Department of Software Engineering and Programming Languages
Institute of Computer Science
Universitätsstr. 6a D-86135 Augsburg

Incremental Model Checking of Recursive Kripke Structures

Jan Finis

Master's Thesis in the Elite Graduate Program Software Engineering

LMU

Universität
Augsburg
University

TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Department of Software Engineering and Programming Languages
Institute of Computer Science
Universitätsstr. 6a D-86135 Augsburg

Incremental Model Checking of Recursive Kripke Structures

Submission Date:	January 23, 2012
First Supervisor:	Prof. Dr. Wolfgang Reif
Second Supervisor:	Prof. Dr. Alexander Knapp
Advisors:	Dr. Ansgar Fehnker, Dr. Gerhard Schellhorn

Software Engineering
Elite Graduate Program

Erklärung

Hiermit versichere ich, dass ich diese Masterbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, 19. Januar 2012 Jan Finis

Acknowledgments

First, I would like to thank my supervisor Prof. Dr. Wolfgang Reif for giving me the chance to write my thesis abroad in the renowned National ICT Australia (NICTA).

Second, I want to thank NICTA for accommodating me during my thesis. Especially, I want to thank my adviser at NICTA, Dr. Ansgar Fehnker, for providing this very interesting topic and for helping me with getting into it.

Third, thanks go to my adviser Dr. Gerhard Schellhorn for providing viable feedback for the thesis. Especially his suggestions about more concise notational conventions made the thesis easier to be read and grasped.

Finally, let me thank my second supervisor Prof. Dr. Alexander Knapp for various intense feedback sessions about the formalization of the algorithms and the proof of their correctness.

Abstract

Static code analysis is a widespread method for automatic bug detection in software. A promising approach for testing whether a program satisfies certain requirements is model checking. Here, a model of the system is built and properties specified in a temporal logic like linear temporal logic (LTL) or computational tree logic (CTL) are checked by thoroughly exploring the state space of the model. The implementation and complexity of the model checking problem depends on the used model. The traditional approach to model checking is using finite transition systems like Kripke structures. While these models are suitable for modeling the local control flow in a function (intra-procedural analysis), they are unable to model the global control flow (inter-procedural analysis) adequately without introducing a very large number of states by "inlining" all function calls. In case of recursion, the resulting model would even become infinitely large, thus totally prohibiting the use of these types of models. A promising replacement for inter-procedural analysis are models which incorporate function calls and are thus able to represent an infinite number of configurations by a finite model. Model checking of such extended models currently is a topic of intense research. A natural extension of Kripke structures are so-called recursive Kripke structures (RKSs) which introduce a call/return concept to Kripke structures. Therefore, RKSs are able to model the global control flow of a program including function calls in a concise and natural way.

In this thesis, a novel algorithm for model checking RKSs with CTL is proposed, which is based on the reduction of the infinite set of configurations of an RKS to a finite set of equivalence classes. The formal basis for the soundness of the algorithm is analyzed and the overall correctness of the algorithm is formally proven. Afterwards, a refinement of the algorithm is presented. Here, incremental checking techniques are introduced in order to drastically reduce the effort and thus increase the performance of the algorithm. The correctness of the incremental refinement is proven as well. The proposed algorithms are implemented into the explicit state model checker XMV, which is used in the static analysis tool Goanna. The concepts for the implementation and optimizations like appropriate data representation and preprocessing are discussed. Finally, a brief evaluation of the implemented algorithms is conducted.

Contents

Glossary

Term	Definition	Defined at page
Boolean and Ternary Logics		
$\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$	The set of boolean logic values	-
$\mathbf{t}$	A boolean / ternary logic *true* value	24
$\mathbf{f}$	A boolean / ternary logic *false* value	24
$\mathbf{m}$	A ternary logic *maybe* value	24
$\mathbb{T} = \mathbb{B} \cup \{\mathbf{m}\}$	The set of ternary logic values	24
Kripke Structures and RKSs		
$\mathcal{K}$	A Kripke structure	11
$\mathbb{P}$	The set of all paths of a Kripke structure	11
$\mathcal{R} = (\mathbb{S}, S^{\text{in}})$	A recursive Kripke structure	17
$\mathbb{R}$	Class of all recursive Kripke structures	19
S	A sub-structure of an RKS	17
S^{in}	Initial sub-structure of an RKS	17
$\mathbb{S}$	Class of all sub-structures of an RKS	17
$\mathbb{S}(\mathcal{R})$	Set of all sub-structures of $\mathcal{R}$	19
$S(c), S(l)$	Sub-structure in which call c or location l is located	19
$S_{\leftarrow c}$	Sub-structure called by call c	19
$S_{\leftarrow \sigma}$	Sub-structure called by call stack σ, that is, called by the topmost call of σ	22
$\mathcal{K}(\mathcal{R})$	The semantic Kripke structure of $\mathcal{R}$	22
$\mathcal{K}(\kappa, S, \theta)$	The associated Kripke structure of S called under context θ. The labels are looked up from κ	48
$\mathcal{K}_{\text{inc}}(\kappa, S, \theta)$	The associated incremental Kripke structure of sub-structure S called under context θ. The labels are looked up from κ	73
$\mathcal{K}(S, \alpha, \eta)$	The associated Kripke structure of sub-structure S using call assumptions α and a context assumption η	96
Kripke Structure Components		
s	A state in a Kripke structure	11
$\mathcal{S}$	Set of states of a Kripke structure	11
I	Set of initial states of a Kripke structure	11
ω	The virtual omega state in $\mathcal{K}$ and $\mathcal{K}_{\text{inc}}$ denoting the successor location of the exit location l^{out}	48
RKS Components		

Term	Definition	Defined at page
l	A location	17
l^{in}	The initial location of a sub-structure	11
l^{out}	The exit location of a sub-structure	17
l^{term}	The virtual termination location of an RKS	22
$l^{\text{succ}}(c)$	The successor location of a call c	19
L	A set of locations of a sub-structure	17
c	A call	17
c^{in}	The virtual initial call which "starts" an RKS	21
C	A set of calls of a sub-structure	17
$l^{\text{in}}(S), l^{\text{out}}(S)$	The initial location / exit location of S	19
$L(S), C(S)$	The set of locations / calls of sub-structure S	19
$L(\mathcal{R}), C(\mathcal{R})$	The set of all locations / calls in all sub-structures of $\mathcal{R}$	19
ν	The call mapping of a sub-structure	17
μ	The labeling function of a sub-structure	17
δ	The transition relation of a sub-structure	17
ν_S, μ_S, δ_S	The call mapping, labeling function, and transition relation of sub-structure S	19
RKS Semantics		
$\sigma = [c_1, \dots, c_n]$	A callstack	21
$\sigma \oplus c$	A callstack with topmost call c	21
$\mathscr{S}(\mathcal{R})$	The set of all callstacks of $\mathcal{R}$	21
(σ, l)	A configuration of an RKS consisting of a call stack σ and a location l	21
$\mathbb{C}$	The set of all configurations of an RKS	22
CTL		
p, q, r	Atomic propositions	-
AP	A set of atomic propositions	-
φ, ψ, χ	CTL Formulae	-
CTL	Class of all CTL Formulae	13
$\ulcorner\varphi\urcorner, \ulcorner\psi\urcorner, \ulcorner\chi\urcorner$	Atomic proposition in the associated Kripke structure denoting the certain validity of φ, ψ, or χ	48
$\ulcorner\varphi?\urcorner, \ulcorner\psi?\urcorner, \ulcorner\chi?\urcorner$	Atomic proposition in the associated Kripke structure denoting that it is unkown whether φ, ψ, or χ is valid	48

Term	Definition	Defined at page
Terms Used by the Model Checking Algorithm		
κ	A knowledge base	36
$\mathbb{K}$	The class of all knowledge bases	36
Ξ	A set of sub-structure call context pairs (S, θ)	40
$\theta = \langle\eta\rangle\tau$	A call context	29
η	A context assumption	29
η_φ	A context assumption for the validity of φ	29
τ	A subformula in a call context	29
θ_φ	A call context with respect to formula φ	29
$\theta_\varphi(\sigma)$	The call context of callstack σ with respect to formula φ	30
$\theta(c, \theta_\varphi)$	The call context of call c with respect to formula φ given that the sub-structure $S(c)$ is called under context θ_φ	35
Θ	The class of all call contexts	29
$\mathit{ica}(\varphi)$	The initial context assumption with respect to φ	37
$\mathit{icc}(\varphi)$	The initial call context with respect to φ	38
$\mathit{cc}(\kappa, c, \theta)$	The call context, as looked up from κ, of the call c which is called under context θ	38
Terms Used by the Basic Approach		
$\gamma_\varphi = (\alpha, \omega)$	A summary for a sub-structure with respect to φ	94
Γ	The class of all summaries for sub-structures	94
π_φ	A summary for an RKS with respect to φ	94
$\pi_\varphi(S) = \gamma_\varphi$	Selection of a summary for sub-structure S with respect to φ	94
Π	The class of all summaries for RKSs	94
$\mathit{ca}(\gamma, c, \eta)$	The context assumption, as looked up in γ, for call c which is called under context assumption η	96
α	The assumption function of a summary	94
ω	The guarantee function of a summary	94
υ	The accumulator mapping in the *split* and *merge* process	102
$\mathcal{K}(S, \alpha, \eta, \varphi)$	The associated Kripke structure of sub-structure S with call assumptions α and context assumption η for checking formula φ	96
$p^{\mathbf{t}}, p^{\mathbf{m}}$	Labels in the associated Kripke structure denoting either the validity of φ or its subformula ψ	96
Miscellaneous		
$\mathbb{N}$	The natural numbers including zero	-
α, β, λ	Locally used, unnamed functions	-

Chapter 1.

Introduction

In this chapter, the field of static analysis using model checking is introduced and the challenges in this field, which form the motivation for this thesis, are discussed. Afterwards, a brief overview over related work in this area of research is given, followed by the description of the scope of this thesis. Finally, the structure of the thesis is summarized.

1.1. Objective & Motivation

The process of writing high-quality software products is becoming more and more challenging due to the increasing program size and time and budget pressure. A key point for ensuring high quality is to analyze the written source code. Basically, this task can be done manually by humans or automatically by software. The manual solution is often performed as code reviews or audits. Here, the problem is that manually looking for bugs in the code is very expensive, time consuming, and error-prone, as a human can always miss some of the bugs. In contrast, automatic code analysis comes virtually for free (once the analysis tools are bought) and usually finds all errors of a certain type without "overlooking" any. The field of *static program analysis* (also called *static code analysis* or merely *static analysis*) consists of all automated techniques for analyzing the source code (or sometimes also the compiled object code) of a program without actually executing this program. Using static program analysis in a software development project usually yields large benefits and is therefore performed in an increasing percentage of software projects. The usage of static program analysis is especially recommended in safety-critical areas where the bugs are intolerable, like in flight control or nuclear power plant control software.

Various techniques exist for performing static analysis. For example, data flow analysis infers propagation of variable values throughout the control flow of the program. This technique could, for example, yield that a null-pointer is propagated to a point in the program where it is dereferenced, which would cause a thread crash and is usually not the intended behaviour of a program. Lately, the technique of using model checking for static program analysis has gained an increasing amount of interest. *Model checking* is a graph exploration technique which is able to verify whether a given property holds in all possible executions of a system represented by a model. The model is usually a state machine like a *Kripke structure* in which states are labeled with certain propositions according to elemental properties which hold in this state. The properties to check for such model are then specified in a temporal logic like computational tree logic (CTL) or linear temporal logic (LTL) which is able to express properties that executions of the system must satisfy. Sometimes, other equivalent temporal property specifications like Büchi automata are used. A model checker is able to determine whether it is guaranteed that such a formula holds

```c
int main(){
    int x, y;
    if(z()){
        x = 5;
    } else {
        y = 2;
    }
    printf("X␣is␣%d",x);
}
```

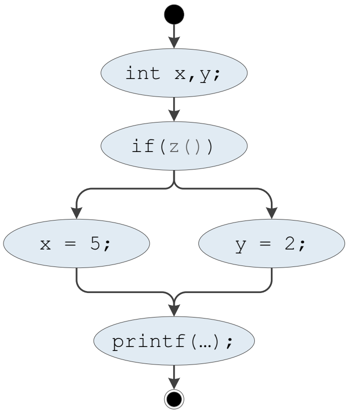


Figure 1.1.: A C function and its corresponding control flow graph

in the system. If it does not hold, a model checker is able to compute a (usually minimal) counter example. Such example usually consists of an execution of the state machine (i.e., a sequence of states) which violates the specified temporal property. For example, the aforementioned example of the null-pointer dereferenciation could also be solved by model checking by constructing a temporal formula which encodes the property "it may never happen that there is an execution in which a null value is assigned to a variable and it is subsequently dereferenced without assigning a non-null value to it inbetween". The model checking problem is seperated into a local and a global one. While the local model checking problem consists of determining whether a temporal property holds in a specific state (often the initial one), the global model checking problem consists of finding the set of all states which satisfy a temporal property.

The usage of model checking for static analysis presumes that a model of the system which is accurate enough to check the desired properties can be built efficiently. The most precise and exhaustive model of a computer program would be to encode its thorough state space into the state machine. However, the state space of a program is usually very large, even for trivial programs. With a (virtually) unbounded memory size, it can even become infinite. Thus, the state space is often no feasible candidate for the static analysis of programs and abstractions have to be made.

An abstraction which is often used for static analysis is the *control flow graph* (CFG) of the program. This directed graph depicts the possible execution paths a program can take. The nodes are for example statements in the code and edges between two nodes A and B depicts that the program execution can go from A to B. Figure 1.1 depicts a C function and its corresponding control flow graph. This model of a function can be used to check certain properties of the local control flow inside the function. Checking local per-function properties is called *intra-procedural* analysis. Although this analysis is already able to find many bugs, it can never find bugs that result from the global control flow of the program. For example, intra-procedural analysis is unable to do null-pointer-dereferenciation checks for global pointer variables that are written in one function and then read in another one. Therefore, it is beneficial to conduct an analysis which is able to analyse the control flow between functions that call each other. This analysis is called *inter-procedural* analysis. Such analysis must work with a model which embodies function calls as well as local intra-procedural control flow. In the example in the figure above, this would mean that the call to function `z()` would have to be represented in the model.

Performing inter-procedural code analysis by model checking is still a challenging field of research. Usual model descriptions like Kripke structures are not suitable for this field anymore, since they fail to represent the function calls efficiently. Therefore, other models which are able to represent the inter-procedural control flow must be used. An example for those models are *recursive Kripke structures* (RKSs), as used in this thesis, which are also often referred to as *recursive state machines* or *hierarchical state machines*[1]. Another prominent model are pushdown automata. The advantage which recursive Kripke structures have compared to other representations is that they represent the control flow graph very naturally and are thus easy to create and interpret.

The Goanna tool [45, 46, 54] is a proof of concept static code analysis tool using model checking. For intra-procedural checking, the tool used the well-known symbolic model checker NuSMV [38]. Because the state space of control flow graphs is rather small, symbolic model checking, which is especially designed for checking very huge state spaces, is not very suitable. The binary decision diagrams (BDDs) used by NuSMV tend to become very large thus making the static analysis process very slow. Therefore, the model checker XMV was developed for the Goanna project. This model checker uses an explicit representation of the state space and yields a performance gain for intra-procedural static analysis compared to NuSMV.

XMV does not support model checking of recursive Kripke structures, yet. This thesis aims to add this support to the model checker, allowing Goanna to perform inter-procedural model checking efficiently. The techniques used are based on the approach of Fehnker et al. [44]. In addition to formalizing and implementing this approach, it is also tried to find optimizations which yield additional performance benefits. These benefits are necessary to allow the inter-procedural analysis of large projects in an acceptable period of time.

1.2. Related Work

Generally, the field of model checking and static program analysis is huge. Therefore, the focus is laid on the narrower field of model checking of recursive state machines and comparable models. All those techniques are based on the basic principles of model checking. For a definitive book about this topic, refer to the book of Baier et al. [11].

Model checking of recursive models is a field of intense study. Many contributions were made for model checking different models with respect to different temporal property specification languages. The most important models are recursive state machines and pushdown automata. Temporal properties are usually specified in CTL* or one of its subsets like CTL, LTL, or EF. Furthermore, Büchi automata are often used instead of LTL or specialized temporal logics are proposed. Finally, also general problems which do not use a certain temporal logic, like reachability, cycle detection, or transitive closure of successors or predecessors, are solved in many contributions. The model checking with respect to a certain temporal logic is often reduced to these problems.

[1] The term "hierarchical" often forbids recursion, thus allowing only non recursive-programs in which the call graph actually forms a directed acyclic graph.

Recursive state machines have been first analyzed by Alur et al. In their first contribution [9], their algorithm is restricted to hierarchical state machines without recursion. They present techniques for solving reachability, cycle detection, and LTL and CTL model checking for those structures. They also find that CTL model checking is PSPACE-complete in the formula depth. Later [5], they add the support of recursive state machines, however, without giving an algorithm for CTL. Benedikt et al.[19] present the first solution for LTL and even CTL* model checking for recursive state machines, which they call unrestricted hierarchical state machines. Finally, Alur et al. and Benedikt et al. combine their findings in a comprehensive paper [2], however, without adding many new results. Their work is the basis for the algorithm of Fehnker et al. [44] and thus also for the algorithms presented in this thesis.

When model checking recursive state machines, a seperation is made between single entry and multiple entry state machines and between single exit and multiple exit state machines. Benedikt et al. [19] show that k entries can be replaced by k machines with single entries and are therefore equally expressive as multiple entry machines, and also equally hard to model check. In contrast, Benedikt et al. show that multiple exit state machines are more expressive than single exit one. They are also exponentially harder to model check in the number of exit states. Finally, Benedikt et al. also compare recursive state machines with pushdown systems and find that a multiple exit recursive state machine is equally expressive as a pushdown automaton, while a single exit state machine is equally expressive as context-free processes. Walukiewicz shows that these are equal to pushdown automata with only one state [72].

Model checking of pushdown automata has been extensively studied by Esparza et al. Here, the semantics of a pushdown automaton are defined via a pushdown system (PDS), which is a usually infinite transition system comprising of the configurations of the pushdown automaton as states. First [21], they propose procedures for reachability, and model checking with LTL, alternation-free μ-calculus, and the logic EF (propositional logic + the EF operator). LTL model checking is done via a transformation of the negated formula to an equivalent Büchi automaton and μ-calculus is model checked via so-called alternating pushdown systems. Finally, they also analyze the compexity: While LTL and the μ-calculus are DEXPTIME-complete in the size of the formula, the logic EF is in PSPACE. In their first contribution, they do not relate their results to static analysis, yet. After some preliminary considerations about model checking in the area of interprocedural dataflow analysis [40], they propose efficient algorithms for model checking pushdown systems with LTL [39]. They use these algorithms in a BDD-based LTL model checker for recursive, boolean programs by modeling the programs as a symbolic pushdown system and using their algorithm on it [42]. This finally results in the static analysis tool Moped [67]. The complexity of pushdown system model checking is further investigated by Bozelli [23]. She proves that the problem is 2EXPTIME-complete for CTL* and EXPTIME-complete for CTL.

Finkel et al. [48] provide a simple algorithm for computing the set of reachable configurations of a pushdown automaton and use this algorithm for model checking these automata with LTL and CTL*. They also show that CTL* model checking stays decidable when allowing regular predicates on the stack content. Hristova et al. [53] propose a quite different approach for LTL model checking of pushdown automata by using evaluation of datalog rules.

A benchmark comparing model checking using pushdown automata with recursive state machines [3] is conducted by Alur et al. They propose algorithms for on-the-fly reachability and cycle detection in recursive state machines. The proposed tool Vera which implements these algorithms is benchmarked against Moped, resulting in a landslide victory for Vera. This highlights the problem which BDD based model checking, when applied to static analysis, has: It is often the case, that the size of the BDDs "explodes".

Vardi et al. investigate in model checking of context free transition systems and prefix recognizable transistion systems. As Benedikt et al. [19] have shown, the former are equivalent to single exit recursive state machines. The approach of Vardi et al. is based on two way alternating automata on infinite trees [71]. They represent the systems as infinite trees with transitions between states specified by finite state automata. The model checking of temporal formulae, which they specify in the μ-calculus [60], LTL [57], and CTL [61], is conducted by an alternating two-way tree automaton navigating the infinite tree. Additionally, they enhance their approach by supporting global model checking in addition to local model checking [64]. Finally, they further formalize their approach introducing tree path automata and add support for fairness constraints and backward modalities [58].

Walukiewicz relates the model checking of pushdown automata using the propositional μ-calculus with winning strategies in games on the graphs of the corresponding pushdown process [72]. Later, he also investigates in model checking of pushdown automata with CTL and EF [73]. He proves that the problem is PSPACE-complete for EF and EXPTIME-complete for CTL.

Recently, Hague et al. [51, 33, 50] have investigated in model checking of higher order pushdown systems in which each element in the stack is a stack itself. This model allows for modelling higher order programs where functions can take other functions as arguments. They use saturation methods and apply these to solve global model checking problems using LTL and branching temporary logics. The model checking problem is solved via the computation of winning regions of a parity game played over the pushdown system. Comparable techniques have also been studied before by Carayol et al. [34] and Cachat [30, 32, 31].

Basler et al. present a very different approach for model checking pushdown systems and boolean programs. They propose a SAT-based method performing bounded model checking, as they argue that most practical PDSs are rather shallow [18]. They reach completeness by using so-called universal summaries, as proposed in [17].

The latest improvements in PDS model checking with CTL come from Song et al. [68]. They reduce the model checking problem to the emptiness problem in alternating Büchi pushdown systems and solve this problem. They argue that their algorithms are more efficient than other existing algorithms for CTL-based PDS model checking, and show this in a benchmark experiment.

The techniques in this thesis work on recursive state machines and are therefore comparable to the techniques of Alur et al., especially since the algorithms in this thesis are based on the ones of Fehnker et al. [44], which are in turn inspired by the work of Alur et al. A big difference between their approach and ours is that we do not construct new state machines during the model checking process implicitly encoding the call context. Instead, we rely on the explicit tracking of call contexts and the gathering of labels in a global knowledge base. While it is not investigated whether this representation yields performance benefits, we think that it is a more natural representation of results which leads to a clearer model checking process. We also conduct a formal proof of the correctness of our algorithm. While also some contributions of Alur et al. contain proof fragments (e.g., [2]), no really complete formal proof is conducted for the correctness of their algorithms. Finally, we also contribute with an incremental refinement of the model checking algorithm which tries to solve the local model checking problem (with respect to the initial configuration) without having to consider the whole recursive state machine. Although the worst case complexity remains the same, it is promising that "real-world" state machines behave benevolent, thus yielding a performance increase. In comparison to model checking techniques for other models like pushdown automata, the advantage of the recursive state machine approaches, when applied to static program analysis, lie in the fact that a recursive state machine is the most intuitive model for the global control flow of a program, resembling almost perfectly the control flow graph (including function calls).

Many of the theoretical concepts have also been implemented in a model checker or whole static analysis suite. Concerning "usual" model checkers for finite state systems, NuSMV [38, 37] is one of the most common ones. It is a symbolic model checker which uses BDDs as an implicit representation of the state space, which allows to model check very large state spaces efficiently. Bebop [13] is also a model checker which uses BDDs but is able to model check boolean programs, which are equivalent to pushdown automata. Therefore, it can be used efficiently for interprocedural dataflow analysis. It is based on an adaption of the reachability algorithm by Reps et al. [65, 66]. Bebop is used as model checker for the SLAM toolkit [14, 12, 15, 16] for static analysis of C code. Further model checkers for pushdown automata are the aforementioned Moped [67] and Mops [36]. Based on Moped, Esarza et al. perform bytecode analysis of Java programs [70, 69]. Model checking of Java programs with pushdown systems was also performed earlier by Obdrzálek [63]. The verification tool Blast [20] is especially suited for model checking C code including pointers and can therefore even be used to model check memory safety (i.e., the dereferenciation of a pointer which points to an invalid memory cell). The SPIN model checker [52] allows specifications only in LTL and transforms them to a Büchi automaton which is used to conduct the checking. It is focused on the efficient verification of asynchronous software systems. The Goanna tool [45, 46, 54] is used for static analysis of C++ programs and is also the target for the implementation in the scope of this thesis. At the beginning, Goanna was using NuSMV as off-the-shelf model checker. Due to the aformentioned BDD-size-explosion, it turned out that using an explicit model checker would yield a huge performance gain over using NuSMV. Such model checker was introduced with XMV, which uses explicit state space representation. The swap to this model checker really yielded a performance increase by orders of magnitude. However, XMV and thus Goanna does not use recursive state machines but uses only model checking of usual Kripke structures on a per-function basis. The aim of this thesis is to enhance XMV to add the missing support for model checking of recursive state machines.

There are many more contributions in this challenging field of research. Some of them focus on special model representations or aim at supporting a wider variety of systems (like concurrent ones). Others propose new logics which are better suited for certain problems. The list of publications in this field is almost endless, so the following further namings are rather a selection instead of a comprehensive list.

Burkart et al. investigate in the model checking of different infinite-state systems, namely context free processes [27], pushdown processes [28], infinite graphs defined by graph grammars [26] and inifinite sequential processes [29].

La Torre et al. [62] extend the concept of the hierarchical state machines of Alur et al. allowing also the labeling of boxes (i.e., calls) with atomic propositions which all locations in the called sub-routine inherit. Therefore, these box labels can be used to model scopes which makes the resulting model more expressive than usual hierarchical state machines. The algorithms of Alur et al. for reachability in recursive state machienes are enhanced by Chaudhuri [35] who proposes algorithms with less than cubic worstcase complexity.

Model checking of concurrent programs is, for example, investigated by Gnesi et al. [49] using communicating UML state machines, Alur et al. [6] using communicating hierarchical state machines, and Esparza et al. [22] using communicating pushdown systems.

The model checking of open systems, that is, system communicating with the environment, as introduced by Vardi et al. [59] using the term *module checking*, is another topic of intense research. The module checking of pushdown systems was, for example, investigated by Bozelli et al. [24], Vardi et al. [10], and Ferrante et al. [47].

The addition of probablism to recursive structures was investigated by Esparza et al. [41, 56] and Brázdil et al. [25] using probablistic pushdown automata, and Etessami et al. [43] using recursive probablistic state machines.

The increase of expressiveness by allowing certain predicates based on the stack content while still staying decidable is, for example, treated by Alur et al. by proposing the logic CaRet [4] which is able to reason about calls and returns and later introducing visibly pushdown automata [7], which tie the pushing or popping of stack symbols only to specific input symbols and allow to specify pre- and postconditions of procedures effectively. Finally, they propose *nested words* [8] as a way for exposing the call-return structure of a program execution and treat programs as finite-state generators of regular languages of these words [1], which allows them to check stronger requirements.

1.3. Scope

In this thesis, the model checking problem of checking recursive Kripke structures with CTL is regarded. The expressiveness of the model is limited on purpose to allow very fast execution of a large number of checks on large models. Precisely, the following three limitations are applied:

- The location labels are fully atomic propositions depending on the syntax of the program, that is, they cannot encode the valuations of variables at that point. For example, the property "variable x is written" can be expressed, while it cannot be expressed that "the value y is written to variable x" unless y is constant.

- The labels in the recursive structure depend only on the location and not on the stack content. Calls also cannot be labeled in the chosen model. Thus, the labels on the locations cannot represent any form of context.

- Only single exit and single entry recursive Kripke structures are regarded, since this form suffices to model inter-procedural control flow naturally.

The basic algorithm in this thesis, which is depicted in Chapter 3, performs global model checking by deciding the validity of a formula in all reachable configurations of an RKS. In contrast, the incremental refinement depicted in Chapter 4 performs local model checking with respect to the initial location of a specified sub-structure (i.e., the entry point of a specific function).

The contribution of this thesis is threefold: It covers theoretical aspects like the formalization and proof of correctness of the model checking algorithm and two practical aspects: The implementation of the model checking algorithm in XMV and the research and implementation of an incremental refinement which is used to increase the performance of the model checker.

The basic task of this thesis is to implement the approach of Fehnker et al. [44] into the model checker XMV and improve it (both by improving the code and by improving the formal concepts of the algorithm) to yield better performance.

The first aspect of this thesis is to implement the aforementioned approach into XMV, thus yielding a model checker which is able to handle recursive Kripke structures.

Before the implementation can start, the approach must be formalized thoroughly. Especially the the sub-tasks *split* and *merge* are only explained very informally in [44]. These sub-tasks must be formalized to be able to implement them. The formalized representation can be found in Appendix A.

After the formalization is accomplished, the formalized algorithm can be implemented into XMV. This includes four main tasks:

1. The input specification language of XMV, which almost exactly resembles the one of NuSMV, has to be enhanced to allow the specification of recursive Kripke structures.

2. A suitable internal data representation for the abstract concepts of the approach has to be found which yields high performance and low memory consumption.

3. The transformation of an input specification to the internal representation has to be conducted. To yield additional performance, optimizations of the input specification, like constant expression propagation and transformation of suitable `case` statements to table-based lookups instead of nested `if` clauses has to be performed.

4. Finally, the formalized approach has to be implemented, working with the model data representation.

An overview over the implementation concepts and challenges can be found in Chapter 5. In addition to the implementation of the algorithm, the non-recursive model checking part of XMV was optimized during this thesis as well, often yielding a ten times speed-up compared to the previous version.

The second part in the scope of this thesis is formal reasoning about the algorithm. The goal is to demonstrate how the algorithm yields semantically correct results. This demonstration is to be formalized by a proof of correctness of the algorithm. To achieve this, the execution semantics of a recursive Kripke structure have to be formalized first. This is done in Chapter 2. Afterwards, the formal basis of the algorithm, namely the context of a call stack and the equivalence of call stacks with respect to their context is elaborated. To ease the formal proof and to make it more intuitive, the formalized algorithm is amended, thus yielding a clearer and better understandable representation of the algorithm. During the presentation of this amended algorithm, examples are given and the formal proof is conducted step by step. This is depicted in Chapter 3.

The final part in the scope of this thesis is the improvement of the model checking algorithm. A refinement has been invented which allows so-called *incremental model checking*. The basic idea behind incremental model checking is to gain performance by not model checking the recursive Kripke structure thoroughly but only the parts which are necessary to yield a correct result for the initial configuration. Here, the challenge lies in maintaining the partly results and deciding which very parts of the structures must be checked to yield a correct result. The incremental refinement is depicted in Chapter 4. This refinement is also implemented into XMV.

1.4. Overview

The rest of this thesis is structured as follows. In Chapter 2, the concepts on which the algorithm is based, like Kripke structures, CTL, recursive Kripke structures, and ternary logic, are defined. Chapter 3 starts with the introduction and reasoning about call contexts and how they partition call stacks into equivalence classes. This chapter contains the proposed model checking algorithm and the proof of its correctness. Chapter 4 presents an incremental refinement of the algorithm from Chapter 3. In addition, the chapter discusses the rationale for the performance increase offered by the incremental refinement and the proof of its correctness. Chapter 5 depicts aspects of the algorithm's implementation into XMV and a brief evaluation. A conclusion and topics for possible future work are shown in Chapter 6. The basic algorithm by Fehnker et al., including the formalization of *split* and *merge*, which is a contribution of this thesis, is presented in Appendix A.

Chapter 2.

Recursive Kripke Structures

This chapter contains the definitions of the structures to be model checked and the definition of the temporal logic CTL which is used to model check these structures. The chapter starts with the definition of Kripke structures on which recursive Kripke structures are based. Afterwards, the syntax of the temporal logic CTL and its semantics are defined. A reduction of the CTL operators to an orthogonal set of operators which suffice for performing the model checking of all operators is conducted. Next, the syntax and semantics of recursive Kripke structures are given. The semantics of CTL for recursive Kripke structures is introduced by defining a semantic Kripke structure corresponding to an RKS and expressing the CTL semantics of RKSs by the semantics of CTL on the corresponding semantic Kripke structure. Finally, ternary logic, which is needed for the model checking algorithm, is introduced.

2.1. Kripke Structures

A Kripke structure, as proposed by Kripke [55], has the semantics of a non-deterministic finite state machine. It consists of a set of states connected by transitions. Each state is labeled with a possibly empty subset of all atomic propositions over which the structure is defined.

Definition 2.1.1 (Kripke structure). *A* Kripke structure $\mathcal{K}$ *over a set of atomic propositions* AP *is defined as* $\mathcal{K} = (\mathcal{S}, I, \delta, \mu)$ *with:*

- *A set of states* $\mathcal{S}$
- *A set of initial states* I *which satisfies* $\emptyset \neq I \subseteq \mathcal{S}$
- *A transition relation* $\delta \subseteq \mathcal{S} \times \mathcal{S}$ *which is left-total, i.e.,* $\forall s \in \mathcal{S} \,.\, \exists s' \in \mathcal{S} \,.\, (s, s') \in \delta$
- *A state labeling* $\mu : \mathcal{S} \to \wp(AP)$

An atomic proposition $p \in AP$ *is also referred to as* label*. A state* s *is said to be* labeled *with* p*, if* $p \in \mu(s)$*.*

A path p *in* $\mathcal{K}$ *is an infinite sequence* $s_0, s_1, s_2, \ldots$ *of states from* $\mathcal{S}$ *which conforms to the transition relation, that is,* $\forall i \in \mathbb{N} \,.\, (s_i, s_{i+1}) \in \delta$*. For each state* $s \in \mathcal{S}$*, there is at least one path which starts at* s*, due to the left-totality of* δ*. The set of all paths of a Kripke structure is denoted by* $\mathbb{P}$*. An* execution *of* $\mathcal{K}$ *is a path* $s_0, \ldots \in \mathbb{P}$ *where* $s_0 \in I$*.*

```c
int main(){
    int x, y;
    if(z()){
        x = 5;
    } else {
        y = 2;
    }
    printf("X␣is␣%d",x);
}
```

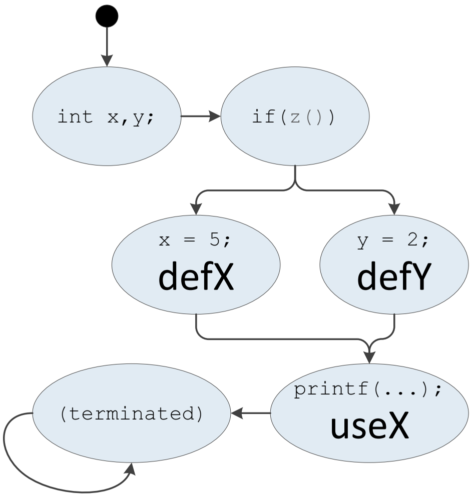


Figure 2.1.: A C function and its corresponding Kripke structure

Kripke structures are used in model checking to represent the behaviour of a system. The labels in a state represent properties which hold in this state. Although the usual definition of Kripke structure restricts S to be finite, Definition 2.1.1 also allows Kripke structures with an infinite number of states, especially since a (finite) recursive Kripke structure may not correspond to a finite Kripke structure but to one with an infinitely large number of states.

In the case of static program analysis, a Kripke structure usually models the control flow graph of a function. In this scenario, safety properties of the code, like that there is no null pointer dereferenciation or that no uninitialized variable is read, are usually checked. Therefore, the states are labeled with specific propositions stating a relevant fact of that state. For example, to check for whether a variable `v` is initialized before it is read, the label `defV` would be inserted at states where `v` is written and `useV` would be inserted where `v` is read.

Example: Figure 2.1 shows an example C function and its corresponding Kripke structure. The structure has been labeled to check whether each local variable is initialized before it is read. The figure depicts the notation that will be used in this work to visualize Kripke structures: The states are circles or ovals and the transitions are arrows. The initial locations are those which have an incoming arrow originating from a black dot. In this example, the set of initial locations I would consist only of the state “`int x,y;`”.

The figure also shows how the usual transformation of a C function to a Kripke structure, which represents the control flow graph of this function, is conducted: Each statement of the function becomes a state and transitions are inserted according to the control flow. For example, in an `if` statement, control can either go to the `then` or to the `else` block, which is depicted by two transitions from the `if(z())` state in the example. If the else block is missing, the second transition must directly go to the next statement after the `if` block. After the final statement, a virtual “terminated” trap state is inserted. This is necessary because the transition relation must be left total. It will be shown later how recursive Kripke structures solve this problem without an extra trap state.

In this example, the small text in the states shows from which statement in the function a state originated. It has no semantic meaning. The bold text, in contrast, shows the labels of a state. For example, the state "`x = 5;`" has a **defX** label, that is $\mu(\text{"}\texttt{x = 5;}\text{"}) = \mathbf{defX}$.

The label **defX** and **useX** are used on states where the variable x is written and read, respectively. So, in this case, the Kripke structure would be defined over the set of atomic propositions: $AP = \{\mathbf{defX}, \mathbf{defY}, \mathbf{useX}, \mathbf{useY}\}$. These labels can be used to test whether the local variables are always initialized before they are used. To perform different checks, other labels would be necessary.

2.2. Computational Tree Logic (CTL)

Computational Tree Logic (CTL) is a temporal logic which is usually used for verification of hardware and software systems. It is based on propositional logic and extends it by adding temporal operators — the so-called path quantifiers. The labeling of a Kripke structure $\mathcal{K}$ represents properties which hold at certain states in the system modeled by the structure. CTL Formulae are used to express temporal properties of the behaviour of that system. They are based on the atomic propositions AP of the labeling of $\mathcal{K}$.

Definition 2.2.1 (CTL Syntax)**.** *A formula φ over a set of atomic propositions AP is recurively defined as follows:*

$$\begin{aligned}\varphi ::= {} & \top \mid \bot \mid p \in AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi \\ & \mid \mathbf{EX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{E}\varphi\mathbf{U}\varphi \mid \mathbf{AX}\varphi \mid \mathbf{AF}\varphi \mid \mathbf{AG}\varphi \mid \mathbf{A}\varphi\mathbf{U}\varphi\end{aligned}$$

The set of all CTL formulae over AP is denoted with CTL^{AP} or merely with CTL *if AP can be inferred from the context.*

The syntax constructs in the first line are the ones of propositional logic. Their semantics are equal to those of the propositional logic, which are not re-narrated here. However, the validity of these formulas is always regarded with respect to a certain state in the Kripke structure. A proposition $p \in AP$ holds in a state if p is a label in that state. $\top$ holds in every state and $\bot$ holds in none.

The remaining operators are path quantifiers. They can be classified into **E**xistential quantifiers, which start with an **E** and for-**A**ll quantifiers, which start with an **A**. **E** quantifiers require that the property holds on at least one path, while **A** quantifiers require that the proprery holds on all possible paths. The second letter of the quantifiers determine on what part of a path the property must hold: **X** states that the property must hold in the ne**X**t state on the path, **F** states that the property holds on at least one state on the path (**F**inally). **G** represents a property which must hold **G**lobally, i.e. in each state on the path. **U** is the only binary path quantifier. It states that the first property must hold **U**ntil the second holds. Note that the second property must appear somewhere on the path, i.e. an infinite path where only ψ holds forever but not χ does not satisfy $\psi\mathbf{U}\chi$. Some definitions use another binary quantifier to state that either the first argument holds forever, or at least until the second holds. This quantifier is often called **W**eak until. Definition 2.2.2 describes the CTL semantics formally.

Definition 2.2.2 (CTL Semantics). *Let φ be a CTL formula and $\mathcal{K} = (\mathcal{S}, I, \delta, \mu)$ a Kripke structure, both over a set of atomic propositions AP. Let $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$ be the set of Boolean logic values. φ is said to hold or be valid in a state s, written as $\mathcal{K}, s \models \varphi$, if the following conditions, based on the structure of φ, hold:*

$$\begin{aligned}
\mathcal{K}, s &\models \top && \Rightarrow \mathbf{t} \\
\mathcal{K}, s &\models \bot && \Rightarrow \mathbf{f} \\
\mathcal{K}, s &\models p && \Rightarrow p \in \mu(s) \\
\mathcal{K}, s &\models \neg\psi && \Rightarrow \mathcal{K}, s \not\models \psi \\
\mathcal{K}, s &\models \psi \vee \chi && \Rightarrow \mathcal{K}, s \models \psi \text{ or } \mathcal{K}, s \models \chi && (\wedge, \rightarrow, \leftrightarrow \text{ analogous}) \\
\mathcal{K}, s_0 &\models \mathbf{EX}\psi && \Rightarrow \exists s_0, s_1, \ldots \in \mathbb{P} \,.\, \mathcal{K}, s_1 \models \psi \\
\mathcal{K}, s_0 &\models \mathbf{EF}\psi && \Rightarrow \exists s_0, s_1, \ldots \in \mathbb{P} \,.\, \exists i \in \mathbb{N} \,.\, \mathcal{K}, s_i \models \psi \\
\mathcal{K}, s_0 &\models \mathbf{EG}\psi && \Rightarrow \exists s_0, s_1, \ldots \in \mathbb{P} \,.\, \forall i \in \mathbb{N} \,.\, \mathcal{K}, s_i \models \psi \\
\mathcal{K}, s_0 &\models \mathbf{E}\psi\mathbf{U}\chi && \Rightarrow \exists s_0, s_1, \ldots \in \mathbb{P} \,.\, \exists k \in \mathbb{N} \,.\, \mathcal{K}, s_k \models \chi \wedge \forall i \in \{0, \ldots, k-1\} \,.\, \mathcal{K}, s_i \models \psi \\
\mathcal{K}, s_0 &\models \mathbf{AX}\psi && \Rightarrow \forall s_0, s_1, \ldots \in \mathbb{P} \,.\, \mathcal{K}, s_1 \models \psi \\
\mathcal{K}, s_0 &\models \mathbf{AF}\psi && \Rightarrow \forall s_0, s_1, \ldots \in \mathbb{P} \,.\, \exists i \in \mathbb{N} \,.\, \mathcal{K}, s_i \models \psi \\
\mathcal{K}, s_0 &\models \mathbf{AG}\psi && \Rightarrow \forall s_0, s_1, \ldots \in \mathbb{P} \,.\, \forall i \in \mathbb{N} \,.\, \mathcal{K}, s_i \models \psi \\
\mathcal{K}, s_0 &\models \mathbf{A}\psi\mathbf{U}\chi && \Rightarrow \forall s_0, s_1, \ldots \in \mathbb{P} \,.\, \exists k \in \mathbb{N} \,.\, \mathcal{K}, s_k \models \chi \wedge \forall i \in \{0, \ldots, k-1\} \,.\, \mathcal{K}, s_i \models \psi
\end{aligned}$$

A formula φ holds for a Kripke structure $\mathcal{K}$, written as $\mathcal{K} \models \varphi$, if it holds in all initial states:

$$\mathcal{K} \models \varphi \Leftrightarrow \forall s \in I \,.\, \mathcal{K}, s \models \varphi$$

The solution set $[\![\varphi]\!]\mathcal{K} \subseteq \mathcal{S}$ of a formula φ in a Kripke structure $\mathcal{K}$ is defined as the set of all states in which φ is valid:

$$[\![\varphi]\!]\mathcal{K} = \{s \in \mathcal{S} \mid \mathcal{K}, s \models \varphi\}$$

Example: Figure 2.2 shows an example Kripke structure $\mathcal{K}$ over the atomic propositions $AP = \{p, q, r, s\}$. To check if a formula holds for this structure, one must check if it holds in the initial location at the top. The following formulae will exemplify the semantics of the operators:

$\mathcal{K} \models \mathbf{AX}p$ The formula holds because all paths from the initial state have a p in the successor state of the initial state.

$\mathcal{K} \models \mathbf{EX}p$ Holds because there exists a path where p holds in the successor state of the initial state. Can also be inferred from the previous example, because of the tautology $\mathbf{AX}\psi \Rightarrow \mathbf{EX}\psi$.

$\mathcal{K} \models \mathbf{A}p\mathbf{U}q$ All paths have p labels in all states until a q label is reached in the state at the bottom.

$\mathcal{K} \models \mathbf{EG}r$ There exists a path, in which r holds in every state.

$\mathcal{K} \not\models \mathbf{EG}p$ There exists no path, in which p holds globally, since all paths reach the state at the bottom which has no p label.

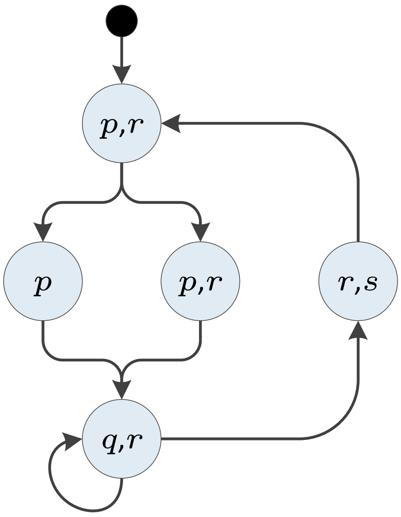


Figure 2.2.: A Kripke structure

$\mathcal{K} \models$ **AF**q All paths eventually enter the bottom state which has a q label.

$\mathcal{K} \not\models$ **AF**s Does not hold, because there exists a path which never reaches the state on the right with the s label. This is the path which stays infinitely in the bottom state.

$\mathcal{K} \models$ **AXAXEG**q Holds, because all paths reach the bottom state after two steps. In this state, there is a path in which q holds globally. This is again the path which stays infinitely in the bottom state.

$\mathcal{K} \models$ **A**p**U**(**E**q**U**s) Holds, because **E**q**U**s holds in the bottom state and p holds on all paths until the bottom state is reached.

A program which determines the validity of a given CTL formula for a given Kripke structure is usually called *model checker*, because it *checks* properties of the Kripke structure, which is usually a *model* of the behaviour of a system. In the case of static program analysis, the model checker checks properties of the control flow of a program.

Example: Again consider the Kripke structure for a C function in Figure 2.1 on page 12. As mentioned, the structure has been labeled to check whether each local variable is initialized before it is read. The label **defX** and **useX** are used on states where the variable `x` is written and read, respectively. To check that `x` is never used before it is initialized, the formula ¬**E**¬**defX U useX** can be used. Informally, the formula states the following: No path may exist (¬**E**) on which `x` is never defined (¬**defX**) before it is read (**U useX**). In the example, this formula does not hold, because the `else` branch does not initialize `x`. Therefore, a static program analysis tool would emit a warning.

2.3. Minimal Representation of CTL

The path quantifiers of CTL are not orthogonal which means that some quantifiers can be expressed through others. In fact, it is possible to use only a minimal set of three quantifiers and express all others by means of them. This has two advantages:

1. A model checker only has to include algorithms to solve the remaining quantifiers instead of all.
2. An efficient algorithm for one quantifier can be used to solve other quantifiers for which only less efficient algorithms exist.
3. Structurally recursive functions have to incorporate only three cases for the path quantifiers and can therefore be specified and proven with less effort.

Hereinafter, the quantifiers are reduced to **EX**, **EG**, and **EU**. The algorithms will only be defined for these. A formula which contains other quantifiers has to be transformed into an equivalent reduced formula. The following equivalences are used for the transformation:

$$\begin{aligned}
\mathbf{EF}\psi &\Leftrightarrow \mathbf{E}\top\mathbf{U}\psi \\
\mathbf{AX}\psi &\Leftrightarrow \neg\mathbf{EX}\neg\psi \\
\mathbf{AF}\psi &\Leftrightarrow \neg\mathbf{EG}\neg\psi \\
\mathbf{AG}\psi &\Leftrightarrow \neg\mathbf{E}\top\mathbf{U}\neg\psi \\
\mathbf{A}\psi\mathbf{U}\chi &\Leftrightarrow \neg((\mathbf{E}\neg\chi\mathbf{U}\neg(\psi \vee \chi)) \vee \mathbf{EG}\neg\chi)
\end{aligned}$$

In addition, the propositional logic operators are also reduced to $\vee$ and $\neg$ with the following equivalences:

$$\begin{aligned}
&\psi \wedge \chi \Leftrightarrow \neg(\neg\psi \vee \neg\chi) \\
&\psi \rightarrow \chi \Leftrightarrow \neg\psi \vee \chi \\
&\psi \leftrightarrow \chi \Leftrightarrow (\psi \rightarrow \chi) \wedge (\chi \rightarrow \psi) \Leftrightarrow \neg(\neg(\neg\psi \vee \chi) \vee \neg(\neg\chi \vee \psi))
\end{aligned}$$

Finally, the constant $\bot$ is also expressed as $\neg\top$ to further reduce the amount of structural cases. Thus, the reduced set of operators which is used for the recursive model checking presented in this thesis is the following:

$$\varphi ::= \top \mid p \in AP \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{EX}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{E}\varphi\mathbf{U}\varphi$$

2.4. Recursive Kripke Structures

The previous section depicted examples how Kripke structures and CTL can be used in static program analysis to check properties of a function. This approach bears the problem that the Kripke structure only represented the local control flow of one single function. Thus, only local intra-procedural properties can be checked. However, many of the problems to be detected by static program analysis need to consider the inter-procedural global control flow.

To consider global control flow, a model must be built which reflects the whole structure of the program, not only a single function. The naïve idea would be to build a Kripke structure for the whole control flow graph of a program by replacing each function call with the states which represent the body of a function. This can be regarded as inlining all functions. The problem is that this Kripke structure would become very large, because n calls of the same function would introduce the states for this function (and all functions called by it) n times. In addition, the resulting Kripke structure would even become infinite in case of recursive calls.

Example: Consider the C program shown in Figure 2.3, which is to be called P. It consists of a main function which calls the second function a two times. The second function calls b. The goal is to create a model of program P to check global properties.

The left side of Figure 2.4 shows the Kripke structures which would result for the three functions, without considering global control flow. The states are named according to the line number in the function (e.g., state **a2** represents the statement in line 2 of function a). The termination state of the functions is denoted with ***t** (with * replaced by the name of the sub-structure). The right side of the figure shows a Kripke structure representing the global control flow of P. As already said, the duplicate call to a results in the duplication of all states of a and all functions called by it.

Due to the resulting state explosion, it cannot be feasible to conduct model checking of real programs, which consist of ten-thousands of functions, by inlining. Therefore, a better model has to be found which can represent the global control flow of a program more compactly. Such a model is the *recursive Kripke structure (RKS)*.

Instead of inlining the calls of other functions into one big structure, the local control flow graphs of each function are kept separate: An RKS $\mathcal{R}$ for a program P is a set of such separate structures, which are called *sub-structures* S of $\mathcal{R}$, and a dedicated initial sub-structure which depicts the "main" function which starts P. Sub-structures can call other sub-structures of $\mathcal{R}$ with special *call* states. A call state c has no labels but instead has a *call target* $\nu(c)$ representing the sub-structure called by this state. States which are no call states are denoted as *locations*. The sub-structures have no termination state anymore. Instead, they have a specific *exit location* l^{out} which has no outgoing transition. Definition 2.4.1 shows the formal definition of an RKS.

Definition 2.4.1 (Recursive Kripke Structures)**.** *A* Recursive Kripke Structure $\mathcal{R}$ *over a set of atomic propositions* AP *is a tuple* $(\mathbb{S}, S^{\text{in}})$*, where* $\mathbb{S}$ *is a set of sub-structures and* $S^{\text{in}} \in \mathbb{S}$ *is the* initial sub-structure*. Each sub-structure* $S \in \mathbb{S}$ *is a tuple* $(L, l^{\text{in}}, l^{\text{out}}, C, \delta, \mu, \nu)$ *with*

- *a set of locations* L*,*
- *an entry location* $l^{\text{in}} \in L$*,*
- *an exit location* $l^{\text{out}} \in L$*,*
- *a set of calls* C*,*
- *a transition relation* $\delta : (L \cup C) \times (L \cup C)$
- *a labeling* $\mu : l \to \wp(AP)$*,*

```
int main(){
    int x = a(2);
    x += 2;
    x = a(x);
    printf("%d",x);
}
```

```
int a(int i){
    int x = 3 + i;
    x = b(x);
    return x;
}
```

```
int b(int i){
    if(i > 3){
        i = 0;
    }
    return i;
}
```

Figure 2.3.: C program with three functions

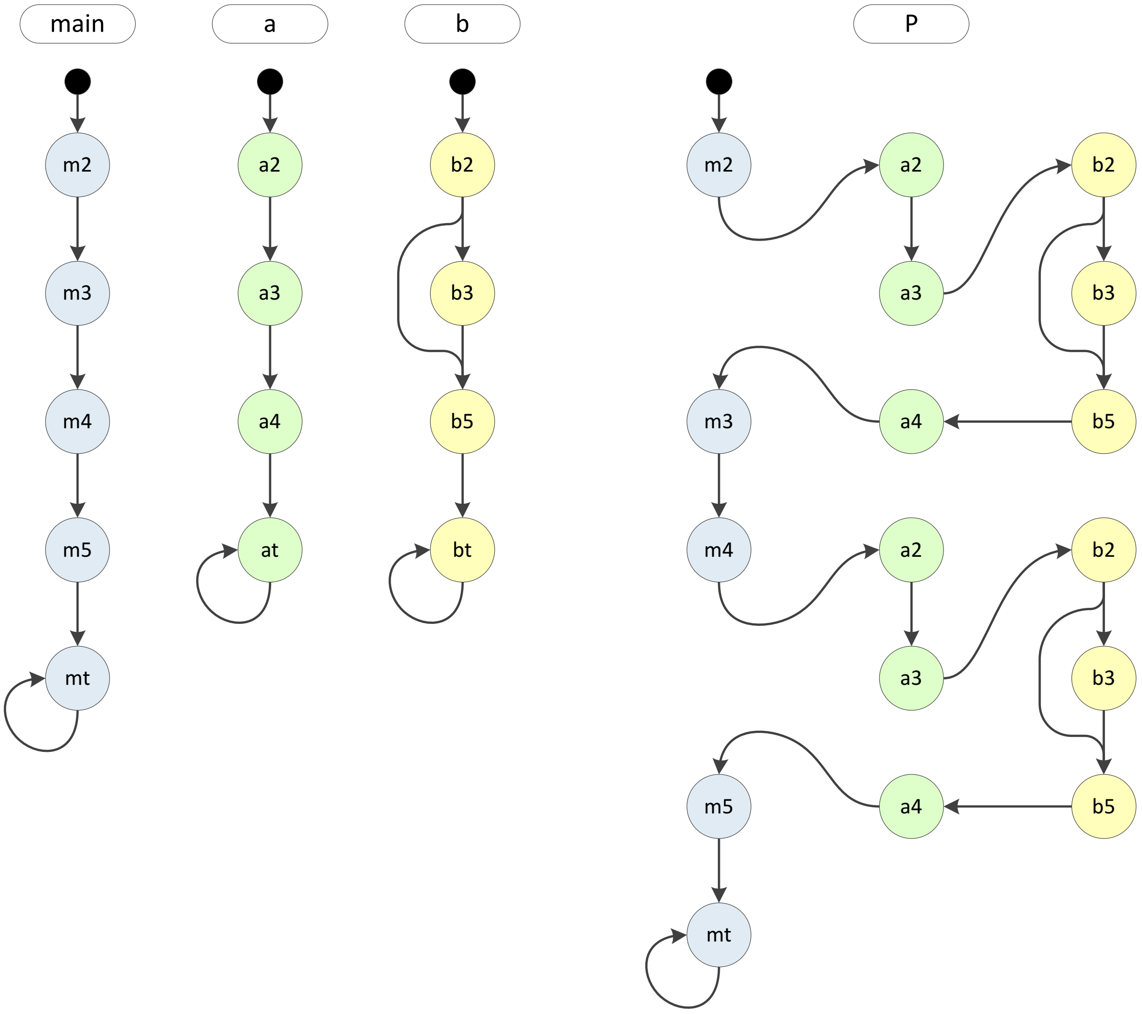


Figure 2.4.: A Kripke Structure for the program from Figure 2.3

- *and a call target mapping* $\nu : C \to \mathbb{S}$

The transition relation δ *satisfies the following requirements:*

$$\forall s \in (L \cup C) \setminus \{l^{\text{out}}\} \,.\, \exists s' \in (L \cup C) \,.\, (s, s') \in \delta \tag{2.1}$$

$$\nexists s \in L \cup C \,.\, (l^{\text{out}}, s) \in \delta \tag{2.2}$$

$$\nexists c, c' \in C \,.\, (c, c') \in \delta \tag{2.3}$$

$$\forall c \in C, l, l' \in L \,.\, ((c, l) \in \delta \wedge (c, l') \in \delta) \Rightarrow l = l' \tag{2.4}$$

The components L*,* C*,* l^{in}*,* l^{out} *of a specific sub-structure* S *are denoted by* $L(S)$*,* $C(S)$*,* $l^{\text{in}}(S)$*, and* $l^{\text{out}}(S)$*, respectively. The components* δ*,* μ*,* ν *of a specific* S *are denoted by* δ_S*,* μ_S*, and* ν_S*, respectively.*

The sets of locations and calls of the sub-structures of $\mathcal{K}$ *are all disjunct:*

$$\forall S \in \mathbb{S} \,.\, L(S) \cap C(S) = \emptyset$$

$$\forall S, S' \in \mathbb{S} \,.\, S \neq S' \Rightarrow (L(S) \cap L(S') = \emptyset \wedge C(S) \cap C(S') = \emptyset \wedge L(S) \cap C(S') = \emptyset)$$

The set of all locations and calls of all sub-structures of $\mathcal{R}$ *is denoted by* $L(\mathcal{R})$ *and* $C(\mathcal{R})$*, respectively:*

$$L(\mathcal{R}) = \bigcup\nolimits_{S \in \mathbb{S}} L(S)$$

$$C(\mathcal{R}) = \bigcup\nolimits_{S \in \mathbb{S}} C(S)$$

The sub-structure in which a specific call c *or location* l *is located is denoted by* $S(c)$ *or* $S(l)$*, respectively. The sub-structure called by a call* c *is denoted by* $S_{\leftarrow c}$ *and the successor location of a call* c *is denoted by* $l^{\text{succ}}(c)$*. :*

$$c \in C(S(c))$$

$$l \in L(S(l))$$

$$S_{\leftarrow c} = \nu_{S(c)}(c)$$

$$(c, l^{\text{succ}}(c)) \in \delta_{S(c)}$$

The set of sub-structures of $\mathcal{R} = (\mathbb{S}, S^{\text{in}})$ *is written as* $\mathbb{S}(\mathcal{R})$*. The class of all RKSs is denoted by* $\mathbb{R}$*.*

As the definition states, a sub-structure S is comparable to a usual Kripke structure with a few modifications. First, the set of states of the sub-structure consists of locations L and calls C and only one initial location l^{in} exists, instead of a set of initial locations. A call $c \in C$ has a specific call target $\nu(c)$. In addition, its transition relation δ is not left total anymore, because no outgoing transition from l^{out} exists. Concerning calls, there are more restrictions for the transition relation: First, no two calls may be behind each other (Equation 2.3). Second, a call must have exactly one successor location (Equation 2.4). Note that these restrictions, together with the restriction that only one initial location per sub-structure may exist, are only introduced to make the model checking algorithm easier to be grasped and proven.

A program P consisting of a set of functions can be transformed into an RKS by transforming the control flow graph of each function to a sub-structure. Each statement which calls another function becomes a call in the respective sub-structure and the call target of that call becomes the sub-structure which was built for the called function. In the graphical notation, squares will be used instead of circles to represent call states. The text in a call-state c will represent the call target sub-structure $S_{\leftarrow c}$. The exit location l^{out} will be depicted by a location which has an outgoing transition to a small filled double circle (◉).

Note that it is often necessary to insert virtual locations to satisfy the definition of an RKS. For example, if the first statement in a function is a function call, then a virtual location has to be generated before that call, because the initial location l^{in} of a sub-structure may not be a call. Likewise, two successive call statements cannot be mapped to two successive calls in the sub-structure. Instead, a virtual location must be inserted between the calls. The only CTL formulae where the insertion of extra locations could yield problems are formulae of the form $\mathbf{EX}\psi$ or $\mathbf{AX}\psi$. However, these formulae are not useful for static program analysis anyway, because it is not exactly defined what "the next step" of a program is.

Example: In contrast to the naïvely constructed large Kripke structure from Figure 2.4, an RKS for program P from Figure 2.3 is to be constructed. Figure 2.5 shows the graphical representation of this RKS.

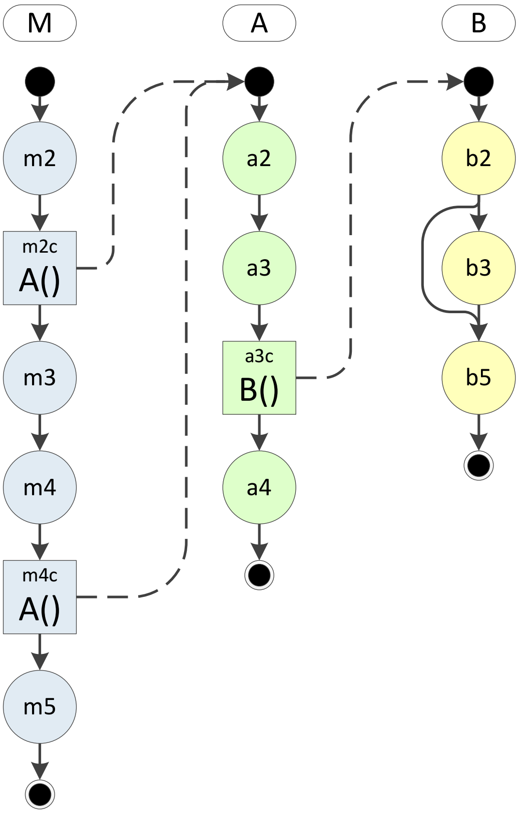


Figure 2.5.: The sub-structures of an RKS modeling the program from Figure 2.3

The RKS $\mathcal{R} = (\{M, A, B\}, M)$ consists of three sub-structures $\mathbb{S} = \{M, A, B\}$, one for each function, with the M sub-structure being the initial one. The function calls are now modeled as call states (rectangles) and a preceding location. The called sub-structure is written in the rectangle. For example, the text `A()` in the call m2c in sub-structure M represents that $\nu_M(\mathsf{m2c}) = S_{\leftarrow \mathsf{m2c}} = A$. The dashed arrows represent this fact. They will be left out in further examples. In this example, the sub-structure M would be specified as:

$$\begin{aligned}
&m = (L, l^{\text{in}}, l^{\text{out}}, C, \delta, \mu, \nu)\\
&L = \{\mathsf{m2}, \mathsf{m3}, \mathsf{m4}, \mathsf{m5}\}\\
&l^{\text{in}} = \mathsf{m2}\\
&l^{\text{out}} = \mathsf{m5}\\
&C = \{\mathsf{m2c}, \mathsf{m4c}\}\\
&\delta = \{(\mathsf{m2}, \mathsf{m2c}), (\mathsf{m2c}, \mathsf{m3}), (\mathsf{m3}, \mathsf{m4}), (\mathsf{m4}, \mathsf{m4c}), (\mathsf{m4c}, \mathsf{m5})\}\\
&\mu = ?\\
&\nu(\mathsf{m2c}) = \nu(\mathsf{m4c}) = A
\end{aligned}$$

Note that μ has not been specified as no specific set of labels to be generated was chosen.

As shown, the definition of an RKS can be used to model interprocedural control flow conveniently without having to inline functions which would possibly cause a state space explosion. The important aspect of RKSs which needs to be specified is the semantics of their execution (i.e., what a path in an RKS is). According to these semantics, the meaning of the validity of CTL formulae when applied to an RKS can be defined and a model checker can be built which checks this validity.

The informal execution semantics of an RKS reflect what can be inferred intuitively from the control flow of the associated program: An RKS can "take a transition" between two locations which are connected by δ, just like a usual Kripke structure. In addition, if it takes a transition from a location to a call, it enters the sub-structure which is the target of that call, starting in the initial location of that sub-structure. Once the RKS reaches an exit location, it returns to the sub-structure which called the current one. This shows that the RKS has to "remember" from which sub-structure the current one was called. Once the initial sub-structure reaches its exit, the RKS "terminates". That is, it gets stuck in a termination trap state.

The formal semantics of the execution of an RKS are quite close to those of a pushdown automaton. The configuration of an RKS at a given step during its execution is a pair (σ, l) of a location l in a sub-structure S and the call stack σ which led to S. The call stack is a stack of calls which were entered to reach the current sub-structure S. A transition from a location to a successor location changes only the location of the configuration. A transition from a location to a successor call pushes that call onto the stack and enters the initial location l^{in} of the called sub-structure. A transition from an exit location l^{out} pops the topmost call from the stack and enters the successor location of that call. Definition 2.4.2 depicts the formal definition of a call stack and an RKS configuration.

Definition 2.4.2 (RKS Configuration). *Let $\mathcal{R} = (\mathbb{S}, S^{\text{in}})$ be an RKS with sub-structures $S = (L, l^{\text{in}}, l^{\text{out}}, C, \delta, \mu, \nu)$. A* call stack *$\sigma = [c_1, \ldots, c_n]$ of $\mathcal{R}$ is a sequence of calls c_i. The first call $c_1 = c^{\text{in}}$ is a special virtual initial call, which calls sub-structure S^{in} and is not contained in any* *sub-structure:*

$$\begin{aligned} S_{\leftarrow c^{\text{in}}} &= S^{\text{in}} && \text{(the call target of } c^{\text{in}} \text{ is } S^{\text{in}}\text{, by definition)} \\ c^{\text{in}} &\notin C(\mathcal{R}) \end{aligned}$$

All further calls c_i with $i \in \{2, \ldots, n\}$ are located in the sub-structure called by c_{i-1}:

$$\forall i \in \{2, \ldots, n\} \,.\, c_i \in C(S_{\leftarrow c_{i-1}})$$

The $\oplus$ operator is used to push a call to the top of a stack:

$$[c_1, \ldots, c_n] \oplus c = [c_1, \ldots, c_n, c]$$

The set of all call stacks of $\mathcal{R}$ is denoted by $\mathscr{S}(\mathcal{R})$. A callstack $\sigma = \sigma' \oplus c$ is said to call a sub-structure *S, written as $S_{\leftarrow\sigma}$, if S is the target of the topmost call c in σ:*

$$S_{\leftarrow \sigma' \oplus c} = S_{\leftarrow c}$$

A configuration *(σ, l) of $\mathcal{R}$ consists of a non-empty call stack $\sigma = [c_1, \ldots, c_n] \in \mathscr{S}(\mathcal{R})$ and a location $l \in L(S_{\leftarrow c_n})$ or it is a special termination configuration $([], l^{\text{term}})$. The location l^{term} is a special virtual location which is not contained in any sub structure:*

$$l^{\text{term}} \notin L(\mathcal{R})$$

The successor location $l^{\text{succ}}(c^{\text{in}})$ of the initial call c^{in} is defined to be l^{term}:

$$l^{\text{succ}}(c^{\text{in}}) = l^{\text{term}}$$

The initial configuration *of an RKS is defined as $([c^{\text{in}}], l^{\text{in}}(S^{\text{in}}))$. The set of all configurations of $\mathcal{R}$ is denoted by $\mathbb{C}(\mathcal{R})$.*

Note that the definition of a call stack already enforces that this stack is reachable, which means that, starting from the initial configuration $([c^{\text{in}}], l^{\text{in}}(S^{\text{in}}))$, each call c_i in the stack is in the set of calls $C(S(c_i - 1))$ of the sub-structure $S(c_i - 1)$ of the call before it.

To define the semantics of CTL formulae, the notion of a path in an RKS $\mathcal{R}$ has to be defined. This is simply a list of configurations which conforms to the informal execution semantics. Instead of defining the path directly, a so-called semantics Kripke structure $\mathcal{K}(\mathcal{R})$ of $\mathcal{R}$ is defined over the configurations of $\mathcal{R}$. Then, a CTL formula holds for $\mathcal{R}$ if it holds in $\mathcal{K}(\mathcal{R})$. The transition relation of $\mathcal{K}(\mathcal{R})$, which is inferred from the informal semantics, defines formally how paths can look like. Definition 2.4.3 shows the CTL semantics for RKSs by first defining an associated Kripke structure and using this structure to define the semantics.

Definition 2.4.3 (CTL Semantics for RKSs). *Let $\mathcal{K} = (\mathbb{S}, S^{\text{in}})$ be an RKS over atomic propositions AP. The* semantics Kripke structure *$\mathcal{K}(\mathcal{R}) = (\mathcal{S}, I, \delta, \mu)$ of $\mathcal{R}$ is a Kripke structure over AP with:*

- *A set of states $\mathcal{S} = \mathbb{C}(\mathcal{R})$*
- *A set of initial states $I = \{([c^{\text{in}}], l^{\text{in}}(S^{\text{in}}))\}$*

- *A labeling function $\mu : \mathcal{S} \times \wp(AP)$ which reflects the labeling of $\mathcal{R}$. The labeling function μ is defined as follows:*

$$\mu((\sigma, l)) = \begin{cases} \mu_{S(l)}(l) & \textit{if } l \in L(\mathcal{R}) \\ \emptyset & \textit{if } l = l^{\text{term}} \end{cases}$$

- *A transition relation $\delta : \mathcal{S} \times \mathcal{S}$. For two configurations (σ, l) and (σ', l'), $((\sigma, l), (\sigma', l'))$ is in δ if and only if one of the following formulae holds:*

$$(l, l') \in \delta_{S(l)} \wedge \sigma = \sigma' \tag{2.5}$$

$$\exists c \in C_{S(l)} \,.\, (l, c) \in \delta_{S(l)} \wedge S_{\leftarrow c} = S(l') \wedge l' = l^{\text{in}}(S(l')) \wedge \sigma' = \sigma \oplus c \tag{2.6}$$

$$\exists c \in C_{S(l')} \,.\, (c, l') \in \delta_{S(l')} \wedge S_{\leftarrow c} = S(l) \wedge l = l^{\text{out}}(S(l)) \wedge \sigma = \sigma' \oplus c \tag{2.7}$$

$$l = l^{\text{out}}(S^{\text{in}}) \wedge l' = l^{\text{term}} \wedge \sigma = [c^{\text{in}}] \wedge \sigma' = [] \tag{2.8}$$

$$l = l' = l^{\text{term}} \wedge \sigma = \sigma' = [] \tag{2.9}$$

A CTL formula φ holds for a configuration (σ, l) of an RKS $\mathcal{R}$, if it holds for that configuration in the associated Kripke structure $\mathcal{K}$:

$$\mathcal{R}, (\sigma, l) \models \varphi \Leftrightarrow \mathcal{K}(\mathcal{R}), (\sigma, l) \models \varphi$$

As already mentioned, the most important part of the semantics Kripke structure is the transition relation δ which defines how executions of that RKS, and thus paths, may look like. Consider a configuration $(\sigma \oplus c, l)$:

1. The RKS can move to a successor location l' of l without altering the call stack (Equation 2.5). The resulting configuration is $(\sigma \oplus c, l')$.

2. The RKS can take a transition from a location to a successor call c' (Equation 2.6). In this case, the call is pushed onto the stack and the location is the initial location of the called sub structure: $(\sigma \oplus c \oplus c', l^{\text{in}}(S_{\leftarrow c'}))$

3. The RKS can return from a sub-structure S if the current location l is the exit location $l^{\text{out}}(S)$ (Equation 2.7). In this case, the topmost call is popped from the stack and the execution continues in the successor location of that call: $(\sigma, l^{\text{succ}}(c))$

4. Finally, when reaching the exit location of the initial sub-structure while only having the initial call c^{in} on the call stack, the RKS enters and gets stuck in the trap termination configuration $([], l^{\text{term}})$ (Equations 2.8 and 2.9).

Example: Consider again the RKS from Figure 2.5 on page 20. Here, the initial configuration is $([c^{\text{in}}], \mathsf{m2})$. From this configuration, the RKS can enter the call m2c, thus yielding the configuration $([c^{\text{in}}, \mathsf{m2c}], \mathsf{a2})$. After going through state a3 with configuration $([c^{\text{in}}, \mathsf{m2c}], \mathsf{a3})$, it would enter call a3c, yielding configuration $([c^{\text{in}}, \mathsf{m2c}, \mathsf{a3c}], \mathsf{b2})$. When taking the "else branch" in function b, the next configuration would be $([c^{\text{in}}, \mathsf{m2c}, \mathsf{a3c}], \mathsf{b5})$. Now, the RKS would return the first time from a function, thus popping of the topmost call a3c from the call stack and entering its successor location a4. The resulting configuration would be

$([c^{\mathrm{in}}, \mathsf{m2c}], \mathsf{a4})$. Since $\mathsf{a4}$ is also an exit location, the execution would return again, yielding $([c^{\mathrm{in}}], \mathsf{m3})$. A complete path in $\mathcal{K}(\mathcal{R})$ starting from the initial configuration looks like this:

$$\begin{array}{llll}
([c^{\mathrm{in}}], \mathsf{m2}), & ([c^{\mathrm{in}}, \mathsf{m2c}], \mathsf{a2}), & ([c^{\mathrm{in}}, \mathsf{m2c}], \mathsf{a3}), & ([c^{\mathrm{in}}, \mathsf{m2c}, \mathsf{a3c}], \mathsf{b2}), \\
([c^{\mathrm{in}}, \mathsf{m2c}, \mathsf{a3c}], \mathsf{b5}), & ([c^{\mathrm{in}}, \mathsf{m2c}], \mathsf{a4}), & ([c^{\mathrm{in}}], \mathsf{m3}), & ([c^{\mathrm{in}}], \mathsf{m4}), \\
([c^{\mathrm{in}}, \mathsf{m4c}], \mathsf{a2}), & ([c^{\mathrm{in}}, \mathsf{m4c}], \mathsf{a3}), & ([c^{\mathrm{in}}, \mathsf{m4c}, \mathsf{a3c}], \mathsf{b2}), & ([c^{\mathrm{in}}, \mathsf{m4c}, \mathsf{a3c}], \mathsf{b3}), \\
([c^{\mathrm{in}}, \mathsf{m4c}, \mathsf{a3c}], \mathsf{b5}), & ([c^{\mathrm{in}}, \mathsf{m4c}], \mathsf{a4}), & ([c^{\mathrm{in}}], \mathsf{m5}), & ([], l^{\mathrm{term}}), \\
([], l^{\mathrm{term}}), & ([], l^{\mathrm{term}}), & ([], l^{\mathrm{term}}), & \ldots
\end{array}$$

The semantics Kripke structure $\mathcal{K}(\mathcal{R})$ of an RKS $\mathcal{R}$ is basically the unrolled Kripke structure of a program, as was shown on the right side of Figure 2.4. As already stated, the goal is to explicitly avoid computations with that Kripke structure, because it may have an infinite number of states in the case of recursive functions: Each recursive call pushes itself onto the call stack, yielding an infinite amount of configurations with growing stack size. Therefore, the semantics Kripke structure is only used to define the intuitive semantics formally. The goal of the algorithm developed in this thesis is to model check an RKS $\mathcal{R}$ with respect to a CTL formula φ without having to construct the semantics Kripke structure $\mathcal{K}(\mathcal{R})$, while still yielding the same results as model checking of $\mathcal{K}(\mathcal{R})$ would have yielded.

2.5. Ternary Logic

The validity of a formula at a given location is expressed in Boolean logic. Either the formula holds, or it does not hold. During the incremental checking of a formula, a third logic value must be added, which states the absence of a safe conclusion. This third value states that a formula *may* hold.

While the Boolean logic values of true and false are denoted by $\mathbf{t}$ and $\mathbf{f}$, respectively, the third *maybe* value is denoted by $\mathbf{m}$.

Ternary logic is consistent with Boolean logic, which means that ternary logic formula will yield the same result as an equivalent Boolean logic formula as long as all propositions are either $\mathbf{t}$ and $\mathbf{f}$. The result only differs in the presence of $\mathbf{m}$ values.

Definition 2.5.1 defines the ternary logic by using only the operators $\vee$ and $\neg$ to define ternary logic. Other operators are inferred from the two operators, as already shown in Section 2.2. In addition, a "join" operator $\bowtie$ is defined which has no equivalent operator in Boolean logic.

Definition 2.5.1 (Ternary Logic). *Let $a, b \in \mathbb{T}$ be ternary logic values with $\mathbb{T} = \mathbb{B} \cup \{\mathbf{m}\}$.*

The binary operator $\vee : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$ is defined as follows:
$a \vee b = \mathbf{t}$ if a or b is $\mathbf{t}$, $a \vee b = \mathbf{f}$ if a and b are $\mathbf{f}$, otherwise $a \vee b = \mathbf{m}$.

The unary operator $\neg : \mathbb{T} \to \mathbb{T}$ is defined as follows:
If a is $\mathbf{t}$, then $\neg a = \mathbf{f}$. If a is $\mathbf{f}$, then $\neg a = \mathbf{t}$. Otherwise, $\neg a = \mathbf{m}$.

The binary operator $\wedge : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$ is defined as $a \wedge b = \neg(\neg a \vee \neg b)$.

The binary operator $\Rightarrow$: $\mathbb{T} \times \mathbb{T} \to \mathbb{T}$ *is defined as* $a \Rightarrow b = \neg a \vee b$.

The binary operator $\Leftrightarrow$: $\mathbb{T} \times \mathbb{T} \to \mathbb{T}$ *is defined as* $a \Leftrightarrow b = (a \Rightarrow b) \wedge (b \Rightarrow a)$.

The binary operator $\bowtie$: $\mathbb{T} \times \mathbb{T} \to \mathbb{T}$ *is defined as follows:*
$a \bowtie b = \mathbf{t}$, *if* a *and* b *are* $\mathbf{t}$, $a \bowtie b = \mathbf{f}$ *if* a *and* b *are* $\mathbf{f}$, *otherwise* $a \bowtie b = \mathbf{m}$.

This definition is consistent with the natural interpretation: If it is unknown if a proposition a holds ($a = \mathbf{m}$), then it is also unknown whether the negation of a holds ($\neg a = \mathbf{m}$). For the logical-or operator $\vee$, having one $\mathbf{t}$ value is enough to decide $a \vee b = \mathbf{t}$, even if the other value is unknown. If one value is unknown and the other one is $\mathbf{f}$, then the result of $a \vee b$ is also unknown, because it depends on the value of the unknown operand.

Chapter 3.

Model Checking Recursive Kripke Structures with CTL

This chapter describes an algorithm for model checking RKSs with CTL. First, the algorithm of Fehnker et al., on which this algorithm is based, is explained. Afterwards the conceptual basis for the algorithm is laid out with the definition and reasoning about call contexts and the knowledge base. Afterwards, the algorithm is defined formally and its correctness is proven. Finally, differences to the algorithm of Fehnker et al. and reasons for those differences are described.

3.1. The Basic Approach

The model checking algorithm presented in this chapter and refined in the next chapter is based on the algorithm of Fehnker et al. [44], which is presented in Appendix A and is hereinafter referred to as *basic approach* or *basic algorithm*. For this thesis, the basic algorithm was amended drastically thus almost yielding a new algorithm. The motivation behind this was to give a more natural representation of the model checking process. However, the amending of the algorithm does not yield any performance increase, it is merely only a conceptual change which helps during the explanation and proof of correctness of the algorithm. Therefore, the algorithm in this chapter was not implemented into XMV. Instead, only the basic algorithm and the incremental refinement presented in the next chapter were implemented.

The basic algorithm and the algorithm presented in this chapter both perform model checking of a nested CTL formula bottom up, that is, starting from the innermost subformulae. For example, the formula **AGEGEF**p would be checked in the order p, **EF**p, **EGEF**p, **AGEGEF**p. This bottom up checking ensures that the model checking results for all subformulae of a formula φ are available when checking φ. For example, when model checking **EGEF**p, the results for **EF**p are already available. The difference between the algorithms lies in how these results are represented. In the basic algorithm, the result of model checking an RKS $\mathcal{R}$ with respect to a formula φ is a new RKS $\mathcal{R}'$ which has φ as an atomic proposition. That is, each location where φ holds is labeled with φ in $\mathcal{R}'$. The different contexts under which the same sub-structure S may be called in $\mathcal{R}$ are incorporated in $\mathcal{R}'$ by multiple copies of S. Thus, the number of sub-structures of $\mathcal{R}'$ is usually greater than the number of sub-structures of $\mathcal{R}$. In contrast, the algorithm presented here uses a global storage of information, the so-called knowledge base κ, instead of "weaving" the results for subformulae into the RKS itself. The knowledge base explicitly maps pairs of locations l and contexts θ_φ with respect to a formula φ to the validity of that formula in location l

when called under context θ_φ. It is believed that this separation of model checking results from the structure of the RKS yields a conceptually clearer algorithm.

In contrast, the step of model checking a formula φ once all results for subformulae of φ are known is similar in the two algorithms. The only changes here are that the temporal operator **EX** is not solved by local model checking and that the algorithmic representation differs, because the validity of subformulae is looked up from κ instead of being represented as labels in the RKS.

3.2. Overview

The algorithm for model checking of RKSs presented here reduces the global problem of solving a formula for the whole RKS to a local problem which solves a formula for only a sub-structure disregarding all other structures except some assumptions about sub-structures which are called by the currently considered sub-structure. This has the advantage, that the underlying semantic Kripke structure of an RKS does not have to be constructed. The construction might be impossible anyway, because the underlying Kripke structure is infinite in the case of recursive calls.

The local checking is conducted by building an associated Kripke structure from a sub-structure S to be checked[1] and the context under which S is called. A labeling is chosen for the built Kripke structure which reflects the labeling of S, the context under which S is called, and the assumptions about sub-structures which are called by the calls in S. The context is represented by information about the validity of the formula to be checked and all its subformulae in the successor location of the exit location of the sub-structure. It is shown that this information is enough to fully specify all relevant information of the context.

Because the information about called sub-structures is incomplete at the beginning, ternary logic must be used where a value of $\mathbf{m}$ specifies yet missing information. If no assumption about a called sub-structure can be given yet, a special $\mathbf{m}$ label is inserted into the associated Kripke structure. The local model checking is repeated iteratively for all sub-structures and yields more and more results in each step, because more and more assumptions about called sub-structures are decided and do no longer have to be labeled $\mathbf{m}$. Once no more new results can be found, the iteration terminates. The validity values obtained during the model checking process are inserted into the so-called knowledge base. If one iteration of the process is treated as a function f which receives a knowledge base and returns a knowledge base with the results from this iteration included, then the algorithm effectively computes the smallest fixed point of f. If still some results are missing once this fixed point is found, they can be decided trivially based only on the temporal operator which is currently checked. Thus, once the algorithm has finished the model checking of a formula φ, the knowledge base includes the validity of φ in each reachable configuration of the RKS. This information is then used for model checking formulae which include φ as subformula, like $\mathbf{EG}\varphi$.

[1] Note that the associated Kripke structure of S may not be mistaken for the semantic Kripke structure for $\mathcal{R}$. While the former is constructed explicitly by the algorithm, the construction of the latter must be avoided.

3.3. Preliminary Considerations

In this section, important concepts which are used by the algorithm, like the call context, how to obtain the context under which a sub-structure is called, and the knowledge base are defined and explained. In addition, important properties about call stacks and their corresponding call contexts are proven, forming the conceptual basis for the correctness of the algorithm.

3.3.1. Call Context

The most important aspect, on which the whole algorithm is built, is the call context. A call context is used while checking a sub-structure locally. It defines in which context the sub-structure is to be checked. The call context in which a sub-structure is checked defines which formulae hold in the successor location of the call calling that sub-structure. Because a sub-structure can be called by more than one call, it is necessary to check such sub-structure under different contexts. A call context is defined as follows:

Definition 3.3.1 (Call Context)**.** *A* call context *θ over a set of atomic propositions AP is defined as a CTL formula over AP where each subformula τ is annotated with a context assumption $\eta \in \mathbb{T}$:*

$$\begin{aligned} \theta &::= \langle \eta \rangle \tau \\ \tau &::= \top \mid p \mid \neg\theta \mid \theta \vee \theta \mid \mathbf{EX}\theta \mid \mathbf{EG}\theta \mid \mathbf{E}\theta\mathbf{U}\theta \end{aligned}$$

where $p \in AP$.

A call context containing a specified associated CTL formula *φ is denoted as θ_φ and satisfies:*

$$\varphi = \theta_\varphi\{\langle \eta \rangle \tau \mapsto \tau\} \tag{3.1}$$

where $X\{A \mapsto B\}$ denotes the replacement of all occurrences of A in X with B.

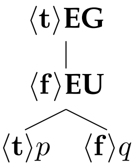
The class of all call contexts is denoted by Θ.

As the definition states, a call context θ_φ is an annotated CTL formula φ. The associated formula φ can be obtained for such context by stripping off the context assumptions, as shown in Equation 3.1. For example:

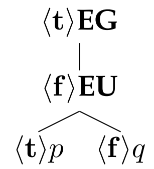
$$\theta_\varphi = \langle \mathbf{t} \rangle \mathbf{EG}(\langle \mathbf{f} \rangle \mathbf{E} \langle \mathbf{t} \rangle p \mathbf{U} \langle \mathbf{f} \rangle q) \Rightarrow \varphi = \mathbf{EG}(\mathbf{E}p\mathbf{U}q)$$

The "inline" notation of the context assumptions is often hard to read. Therefore, a tree-like structure can be used to visualize a call context:

$\langle \mathbf{t} \rangle \mathbf{EG}(\langle \mathbf{f} \rangle \mathbf{E} \langle \mathbf{t} \rangle p \mathbf{U} \langle \mathbf{f} \rangle q) =$



As mentioned, a formula context for a sub-structure S represents a set of formulae which hold in the successor location of a call calling S: All subformulae annotated with $\langle \mathbf{t} \rangle$ hold, those which are annotated with $\langle \mathbf{f} \rangle$ do not hold. If a context contains $\langle \mathbf{m} \rangle$ assumptions, then it is yet unknown whether the corresponding formula holds. The aforementioned context



states that the formulae p and $\mathbf{EG}(\mathbf{E}p\mathbf{U}q)$ hold, while $\mathbf{E}p\mathbf{U}q$ and q do not hold.

Note that not all contexts are reasonable. For example, the context $\langle\mathbf{f}\rangle\mathbf{E}\langle\mathbf{t}\rangle p\mathbf{U}\langle\mathbf{t}\rangle q$ would imply that the proposition q holds in the successor location but $\mathbf{E}p\mathbf{U}q$ does not, which is a contradiction. Also, the context $\langle\mathbf{f}\rangle\top$ is unreasonable since $\top$ always holds. Such impossible contexts are called *inconsistent*. Although the algorithm sometimes produces such inconsistent contexts as intermediary results, the overall result is still correct.

Each reachable call stack of an RKS can be assigned an associated call context which is defined as follows:

Definition 3.3.2 (Call context of a call stack). *Let $\sigma = \sigma' \oplus c$ be a non-empty call stack of an RKS $\mathcal{R}$ and let φ be a CTL formula. The predicate valid(φ, σ) : $\mathrm{CTL} \times \mathscr{S}(\mathcal{R}) \to \mathbb{B}$ denotes the validity of φ in $(\sigma', l^{\mathrm{succ}}(c))$:*

$$\mathit{valid}(\varphi, \sigma' \oplus c) = (\mathcal{R}, (\sigma', l^{\mathrm{succ}}(c)) \models \varphi)$$

The call context $\theta_\varphi(\sigma)$ of σ for φ *is defined recursively over the structure of φ as follows:*

$$\begin{array}{lcl}
\theta_{\varphi\equiv\top}(\sigma) & = & \langle\mathbf{t}\rangle\top \\
\theta_{\varphi\equiv p}(\sigma) & = & \langle \mathit{valid}(\varphi,\sigma)\rangle p \\
\theta_{\varphi\equiv\neg\psi}(\sigma) & = & \langle \mathit{valid}(\varphi,\sigma)\rangle \neg\theta_\psi(\sigma) \\
\theta_{\varphi\equiv\psi\vee\chi}(\sigma) & = & \langle \mathit{valid}(\varphi,\sigma)\rangle (\theta_\psi(\sigma) \vee \theta_\chi(\sigma)) \\
\theta_{\varphi\equiv\mathbf{EG}\psi}(\sigma) & = & \langle \mathit{valid}(\varphi,\sigma)\rangle \mathbf{EG}\theta_\psi(\sigma) \\
\theta_{\varphi\equiv\mathbf{EX}\psi}(\sigma) & = & \langle \mathit{valid}(\varphi,\sigma)\rangle \mathbf{EX}\theta_\psi(\sigma) \\
\theta_{\varphi\equiv\mathbf{E}\psi\mathbf{U}\chi}(\sigma) & = & \langle \mathit{valid}(\varphi,\sigma)\rangle \mathbf{E}\theta_\psi(\sigma)\mathbf{U}\theta_\chi(\sigma)
\end{array}$$

The definition formalizes what was already stated informally above: A call context $\theta_\varphi(\sigma)$ for a formula φ and a call stack $\sigma = \sigma' \oplus c$ describes which subformulae of φ hold in the configuration which is reached after the topmost call c "returns".

Note that by definition, the call context of a call stack never contains $\mathbf{m}$ assumptions, because the predicate valid can either be $\mathbf{t}$ or $\mathbf{f}$. The algorithm however, will use contexts with $\mathbf{m}$ assumptions as intermediary results, when the validity in the successor location is not known, yet.

Example: Consider the RKS $\mathcal{R} = (\{A, B\}, A)$ shown in Figure 3.1, which is defined over the set of atomic propositions $AP = \{p, q\}$. The sub-structure B is called with three different calls from A and one recursive call inside B itself. Thus, the possible call stacks "ending" at B without any recursion are $\mathscr{S}_{\mathsf{NoRec}} = \{[c^{\mathrm{in}}, \mathsf{a2}], [c^{\mathrm{in}}, \mathsf{a3}], [c^{\mathrm{in}}, \mathsf{a4}]\}$. In addition, any of the call-stacks from $\mathscr{S}_{\mathsf{NoRec}}$ with an arbitrary number of recursive calls b2 appended to it is a call stack of $\mathcal{R}$, like $[c^{\mathrm{in}}, \mathsf{a3}, \mathsf{b2}, \mathsf{b2}]$. Therefore, the set of call stacks $\mathscr{S}(\mathcal{R})$ of $\mathcal{R}$ is infinite.

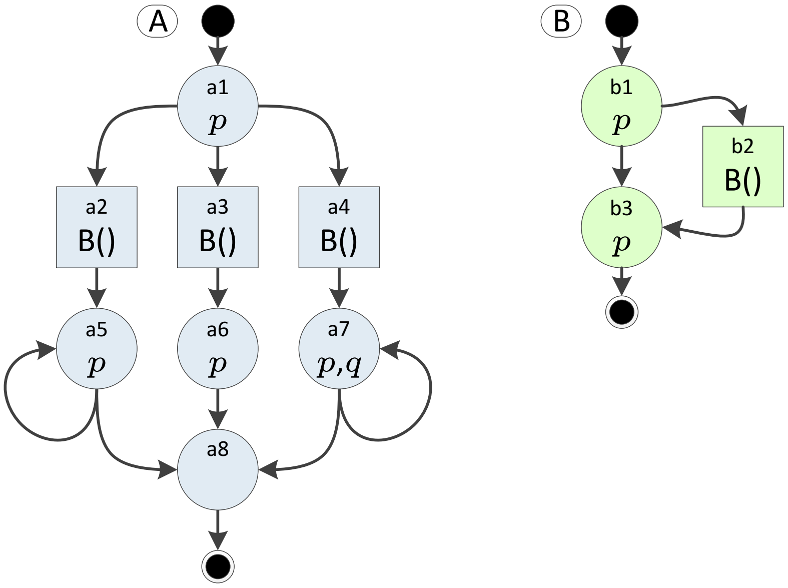


Figure 3.1.: An RKS for exemplifying call contexts

However, the number of different call contexts of stacks in $\mathscr{S}(\mathcal{R})$ with respect to a given formula φ is finite.

For example, in case of $\varphi = \mathbf{EG}p$, the formula is certainly satisfied in the locations a5 and a7, because they are labeled with p and have a self-loop. Therefore, the callstacks $[c^{\text{in}}, \mathsf{a2}]$, $[c^{\text{in}}, \mathsf{a4}]$ certainly yield the call context in which p and $\mathbf{EG}p$ are both tagged $\mathbf{t}$:

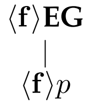
$$\begin{array}{c}\langle \mathbf{t} \rangle \mathbf{EG} \\ | \\ \langle \mathbf{t} \rangle p\end{array}$$

In contrast $\mathbf{EG}p$ does not hold in the successor location a6 of call a3, because it has no self loop and its only successor a8 is not labeled with p. However, a6 is labeled with p, so this subformula receives a $\mathbf{t}$ label. Therefore, the call context of callstack $[c^{\text{in}}, \mathsf{a3}]$ is the following:

$$\begin{array}{c}\langle \mathbf{f} \rangle \mathbf{EG} \\ | \\ \langle \mathbf{t} \rangle p\end{array}$$

Any call stack having a recursive call of B would have a call context in which p is labeled $\mathbf{t}$, since the successor location b3 of the recursive call b2 is labeled with p. The validity of $\mathbf{EG}p$ would be based on which of the calls in A was chosen at the beginning of the stack. For example, the call stack $[c^{\text{in}}, \mathsf{a3}, \mathsf{b2}, \mathsf{b2}]$ would have a context in which $\mathbf{EG}p$ is tagged $\mathbf{t}$, while the stack $[c^{\text{in}}, \mathsf{a2}, \mathsf{b2}, \mathsf{b2}, \mathsf{b2}, \mathsf{b2}]$ would have an $\mathbf{f}$-tagged context with respect to $\mathbf{EG}p$. The recursive stacks are equal to the non-recursive ones and only the two aforementioned contexts exist for $\varphi = \mathbf{EG}p$.

Finally there is the call stack $[c^{\text{in}}]$ which targets A. No labels are assigned to the successor location of this call stack, which is the virtual termination location l^{term}. Therefore p and thus $\mathbf{EG}p$ are not satisfied here, yielding the following call context:



The example shows that the number of different call contexts with respect to a formula is usually much smaller than the amount of call stacks in which a sub-structure is called. The number is even bounded by the depth and arity of the formula, since a context can only vary in each context assumption being either $\mathbf{t}$ or $\mathbf{f}$[2]. For example, there are only $2^4 = 16$ different call contexts for the formula $\mathbf{E}(\mathbf{EG}p)\mathbf{U}q$, since these contexts contain four context assumptions which can be either true or false. This especially implies that the number of call contexts can never be infinite, as opposed to the number of call stacks which usually is. Therefore, the effort for checking a sub-structure with respect to all the call contexts is less than the (possibly infinite) effort of checking it for all call stacks. Therefore, the concept of using call contexts for checking sub-structures locally while achieving a globally correct result is the key of the algorithm. To show that this approach is valid, it has to be shown that two call stacks are equivalent if their call contexts match:

Lemma 3.3.3 (Call stacks context equivalence). *Let $\mathcal{R}$ be an RKS, φ be a CTL formula and let $\sigma \oplus c$ and $\sigma' \oplus c'$ be two non-empty, reachable call stacks of $\mathcal{R}$ which both call the same sub-structure $S = S_{\leftarrow c} = S_{\leftarrow c'}$ and have the same call context $\theta_\varphi = \theta_\varphi(\sigma \oplus c) = \theta_\varphi(\sigma' \oplus c')$ with respect to φ.*

In this case, the two call stacks are equivalent with respect to the validity of φ for all location $l \in L(S)$:

$$\forall \sigma \oplus c, \sigma' \oplus c' \in \mathscr{S}(\mathcal{R}) \,.\, \theta_\varphi(\sigma \oplus c) = \theta_\varphi(\sigma' \oplus c') \wedge S_{\leftarrow c} = S_{\leftarrow c'} \Rightarrow$$
$$(l \in L(S_{\leftarrow c}) \,.\, (\mathcal{R}, (\sigma \oplus c, l) \models \varphi) \Leftrightarrow (\mathcal{R}, (\sigma' \oplus c', l) \models \varphi))$$

The lemma justifies that it is correct to check a sub-structure only under all call contexts of call stacks which call it instead of checking it under all possible call stacks. As shown in the lemma, the equivalence requires that the call context of the stacks and the sub-structure $S_{\leftarrow\sigma}$ called by the stacks must be equal. Using this equivalence, an equivalence relation can be defined and call stacks can be partitioned into equivalence classes. A feasible representation of an equivalence class of call stacks σ is a pair (S, θ) of the call context $\theta = \theta(\sigma)$ and the call target $S = S_{\leftarrow\sigma}$ of σ. This pair is the main structure which is checked at once by the algorithm.

Definition 3.3.4 (Call stack equivalence). *Let φ be a CTL formula and $\mathcal{R}$ an RKS. Two call stacks σ and σ' of $\mathcal{R}$ are* equivalent *with respect to φ, written as $\sigma \sim_\varphi \sigma'$, if*

$$\sigma \sim_\varphi \sigma' \Leftrightarrow \theta_\varphi(\sigma) = \theta_\varphi(\sigma') \wedge S_{\leftarrow\sigma} = S_{\leftarrow\sigma'}$$

Using this equivalence, the equivalence class of a call stack σ with respect to φ is defined as:

$$[\sigma]_\varphi = \{\sigma' \in \mathscr{S}(\mathcal{R}) \mid \sigma \sim_\varphi \sigma'\}$$

A pair (S, θ) of a sub-structure S and a call context θ is called context pair. *The context pair of a call-stack σ with respect to a formula φ is defined as $(S_{\leftarrow\sigma}, \theta_\varphi(\sigma))$. This pair uniquely identifies the equivalence class $[\sigma]_\varphi$ of σ, that is:*

$$\forall \sigma' \in \mathscr{S}(\mathcal{R}) \,.\, (S_{\leftarrow\sigma}, \theta_\varphi(\sigma)) = (S_{\leftarrow\sigma'}, \theta_\varphi(\sigma')) \Leftrightarrow \sigma' \in [\sigma]_\varphi$$

[2] By definition, also $\mathbf{m}$ is a possible context assumption. However, this value only represents missing information. When all information is available, no contexts with $\mathbf{m}$ assumptions exist.

The algorithm will not use any call stacks directly. Instead, it will only infer reachable context pairs which are to be labeled. Since a context pair represents the equivalence class of a call stack with respect to a formula φ and, due to Lemma 3.3.3, all call stacks in that class are equivalent with respect to the validity of φ in the target sub-structure, it is reasonable to use the context pair as surrogate for equivalent call stacks.

Example: Consider again the RKS $\mathcal{R} = (\{A, B\}, A)$ shown in Figure 3.1 on page 31 with the formula $\varphi = \mathbf{EG}p$. As shown in the previous example, the three call contexts for that formula are the following:

$$\theta_1 = \begin{array}{c} \langle\mathbf{f}\rangle\mathbf{EG} \\ | \\ \langle\mathbf{t}\rangle p \end{array} \qquad \theta_2 = \begin{array}{c} \langle\mathbf{t}\rangle\mathbf{EG} \\ | \\ \langle\mathbf{t}\rangle p \end{array} \qquad \theta_3 = \begin{array}{c} \langle\mathbf{f}\rangle\mathbf{EG} \\ | \\ \langle\mathbf{f}\rangle p \end{array}$$

However, the call stacks must also target the same sub-structure to be equivalent. Therefore, it must be distinguished between stacks that target A and those that target B. In this case, there is no call stack with the same context which targets different sub-structures. Therefore, only three equivalence classes with respect to formula $\varphi = \mathbf{EG}p$ exist:

- Call stacks with context pair (B, θ_1). These are the stacks which include the call a3, because $\mathbf{EG}p$ is **f** at that call only:

$$\big[[c^{\text{in}}, \mathsf{a3}]\big]_{\varphi} = \{[c^{\text{in}}, \mathsf{a3}], [c^{\text{in}}, \mathsf{a3}, \mathsf{b2}], [c^{\text{in}}, \mathsf{a3}, \mathsf{b2}, \mathsf{b2}], \ldots\}$$

- Call stacks with context pair (B, θ_2). These are the stacks which originate from calls in A except a3, because $\mathbf{EG}p$ must be **t** at the successor location of these calls:

$$\big[[c^{\text{in}}, \mathsf{a2}]\big]_{\varphi} = \begin{Bmatrix} [c^{\text{in}}, \mathsf{a2}], [c^{\text{in}}, \mathsf{a2}, \mathsf{b2}], [c^{\text{in}}, \mathsf{a2}, \mathsf{b2}, \mathsf{b2}], \ldots, \\ [c^{\text{in}}, \mathsf{a4}], [c^{\text{in}}, \mathsf{a4}, \mathsf{b2}], [c^{\text{in}}, \mathsf{a4}, \mathsf{b2}, \mathsf{b2}], \ldots \end{Bmatrix}$$

- Call stacks ending at A. Since no call explicitly calls A, the only call stack which targets A is the initial one $[c^{\text{in}}]$. This call stack has the context θ_3, because all subformulae are **f** at the virtual termination location.

$$\big[[c^{\text{in}}]\big]_{\varphi} = \{[c^{\text{in}}]\}$$

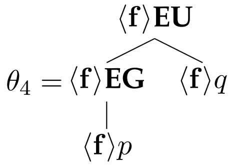
Thus, in the case of $\varphi = \mathbf{EG}p$, there would only be the three context pairs $(B, \theta_1), (B, \theta_2)$, (A, θ_3) to be model checked.

When checking nested formulae, the amount of equivalence classes usually increases (bounded by the formula depth and arity). For example, consider the nested formula $\varphi = \mathbf{E}(\mathbf{EG}p)\mathbf{U}q$. In this case, the formula itself and the proposition q are satisfied in the successor of call a4 but not in any other call in A. Therefore, the following call contexts for the three calls in A would exist:

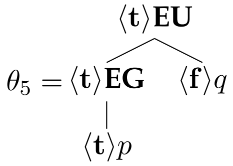
$$\theta_1 = \begin{array}{ccc} & \langle\mathbf{f}\rangle\mathbf{EU} & \\ \swarrow & & \searrow \\ \langle\mathbf{t}\rangle\mathbf{EG} & & \langle\mathbf{f}\rangle q \\ | & & \\ \langle\mathbf{t}\rangle p & & \end{array} \quad \theta_2 = \begin{array}{ccc} & \langle\mathbf{f}\rangle\mathbf{EU} & \\ \swarrow & & \searrow \\ \langle\mathbf{f}\rangle\mathbf{EG} & & \langle\mathbf{f}\rangle q \\ | & & \\ \langle\mathbf{t}\rangle p & & \end{array} \quad \theta_3 = \begin{array}{ccc} & \langle\mathbf{t}\rangle\mathbf{EU} & \\ \swarrow & & \searrow \\ \langle\mathbf{t}\rangle\mathbf{EG} & & \langle\mathbf{t}\rangle q \\ | & & \\ \langle\mathbf{t}\rangle p & & \end{array}$$

The context θ_1 is for the call stack $[c^{\text{in}}, \mathsf{a2}]$, since $\mathbf{EG}p$ holds in a5 but neither q nor $\mathbf{E}(\mathbf{EG}p)\mathbf{U}q$. The second context θ_2 is for $[c^{\text{in}}, \mathsf{a3}]$, since no subformula except p holds in a6. Finally, the call stack $[c^{\text{in}}, \mathsf{a4}]$ would yield call context θ_3, because all subformulae are satisfied in a7.

The initial call stack $[c^{\text{in}}]$ yields again a call context in which no proposition or subformula is valid:


$\langle\mathbf{f}\rangle\mathbf{EU}$
$\theta_4 =$
$\langle\mathbf{f}\rangle\mathbf{EG}$
$\langle\mathbf{f}\rangle q$
$\langle\mathbf{f}\rangle p$

Finally, there are call stacks which include recursive calls from b2. In the case of $\varphi = \mathbf{EG}p$, these call stacks were in the same equivalence class as the ones without recursive calls. This is still true for call stacks which include the call a2 or a3. However, call stacks including a4 are different. In the context θ_3 of the non-recursive stack $[c^{\text{in}}, \mathsf{a4}]$, all subformulae hold. In all recursive call stacks originating from this one, like $[c^{\text{in}}, \mathsf{a4}, \mathsf{b2}]$, the proposition q is no longer valid, since b3, the successor location of b2, is not labeled q. All other subformulae are still valid, yielding the following call context:


$\langle\mathbf{t}\rangle\mathbf{EU}$
$\theta_5 =$
$\langle\mathbf{t}\rangle\mathbf{EG}$
$\langle\mathbf{f}\rangle q$
$\langle\mathbf{t}\rangle p$

Thus, there are 5 equivalence classes with respect to $\varphi = \mathbf{E}(\mathbf{EG}p)\mathbf{U}q$, which are identified by the pairs $(A, \theta_4), (B, \theta_1), (B, \theta_2), (B, \theta_3), (B, \theta_5)$. This is still a lot less than the possible upper bound which would be $2 \times 2^4 = 32$ (2 sub-structures and 2^4 possible contexts, since each of the 4 context assumptions can be either $\mathbf{t}$ or $\mathbf{f}$).

Proof of Lemma 3.3.3. The proof is conducted by structural induction over the formula φ. For each structural case, it must be shown that the validity of the formula only depends on the call context and not on other assumptions made about the call stack. Because the call stacks have the same call context, the same formulae hold in the respective successor locations $l^{\text{succ}}(c)$ and $l^{\text{succ}}(c')$ of their topmost calls.

The base cases $p \in AP$ and $\top$ trivially do not depend on the call stack at all. The operators $\neg$ and $\vee$ do not "look further down paths in the RKS". Therefore, the lemma trivially holds for them, given that the lemma holds for subformulae (which is assumed due to the induction hypothesis). The only formulae which can really introduce an influence of the call stack at all are the path quantors **EG**, **EU**, and **EX**:

Case $\varphi \equiv \mathbf{EX}\psi$: In this case, only the validity of φ in (σ, l^{out}) introduces a dependency on the context; other locations only depend on the validity of ψ in the successor location for which the lemma is true (induction hypothesis). For the exit location, the validity depends on the validity of ψ in l^{succ} which is known from θ_ψ and equal for both call stacks.

Case $\varphi \equiv \mathbf{EG}\psi$: For this operator, two cases must be considered:

1. A path exists which satisfies ψ in each state and never reaches the exit location l^{out} and thus also not l^{succ}. Such path trivially does not introduce a context dependency.
2. No such path exists. In this case, $\mathbf{EG}\psi$ holds only if there exists a path which goes through l^{succ} and satisfies ψ in each location. The following tautology is used to show that, given that the validity of $\mathbf{EG}\psi$ in l^{succ} is known, the validity of $\mathbf{EG}\psi$ in any location can be deduced without "looking into the call stack" and only using information from the call context:

   $$\varphi \equiv \mathbf{EG}\psi \Leftrightarrow \mathbf{E}\psi\mathbf{U}(\mathbf{EG}\psi) \equiv \mathbf{E}\psi\mathbf{U}\varphi$$

   The formula $\mathbf{EG}\psi$ holds exactly if there is a path where ψ is satisfied until $\mathbf{EG}\psi$ itself is known to be satisfied. It is already known from the call context θ_φ whether $\mathbf{EG}\psi$ is satisfied in l^{succ} for both $l^{\mathrm{succ}}(c)$ and $l^{\mathrm{succ}}(c')$. Thus, the formula holds if there exists a path where ψ holds until l^{succ} is reached and ψ holds in l^{succ} ($\mathbf{E}\psi\mathbf{U}\varphi$). States after l^{succ} do not have to be considered.

Case $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$: For this operator, three cases must be considerd.

1. Equal to $\varphi \equiv \mathbf{EG}\psi$, case 1.
2. Similar to $\varphi \equiv \mathbf{EG}\psi$, case 2. In this case the tautology

   $$\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi \Leftrightarrow \mathbf{E}\psi\mathbf{U}(\mathbf{E}\psi\mathbf{U}\chi) \equiv \mathbf{E}\psi\mathbf{U}\varphi$$

   is used, which is equivalent to the one for $\mathbf{EG}\psi$.
3. A path exists which goes through l^{out} but already satisfies $\psi\mathbf{U}\chi$ before it reaches the location. This path also does not introduce a dependency to the call stack.

□

The lemma is the basis for the main idea of the algorithm: To model check a formula φ, it is sufficient to check all reachable context pairs (S, θ) instead of checking all call stacks. For a formula φ, the goal of the labeling algorithm is to label all locations $l \in L(S)$ in all reachable context pairs (S, θ_φ) with either $\mathbf{t}$ or $\mathbf{f}$ depending on whether φ holds in l when S is called under a call stack having θ_φ as call context.

Since call stacks with the same call context yield an equal labeling of locations in the called sub-structure, they especially also yield an equal labeling for the successor locations of calls in that sub-structure. Therefore, it is handy to define the context which a call has when its sub-structure is called under another call context:

Definition 3.3.5 (Call context of call)**.** *Let φ be a CTL formula, and S a sub-structure. Let θ_φ be a call context for S with respect to φ. Let $c \in C(S)$ be a call in S. Let σ be an arbitrary call stack which targets S (i.e., $S_{\leftarrow\sigma} = S$) and has call context θ_φ (i.e., $\theta_\varphi(\sigma) = \theta_\varphi$) The call* context of c with respect to φ under context θ_φ, *written as $\theta(c, \theta_\varphi)$, is defined as follows:*

$$\theta(c, \theta_\varphi) = \theta_\varphi(\sigma \oplus c)$$

The upper definition shows that a context of a call c can be inferred given that a context under which the sub-structure in which c resides is known. Although it theoretically allows to infer new contexts from existing ones, it is non operational since it requires to find a call stack σ with a specific context.

3.3.2. Knowledge Base

The algorithm decides formulae incrementally: Whenever it is able to decide whether a formula φ holds in a location $l \in L(S)$ under a context pair (S, θ_φ), it stores this information in a data structure — the so-called knowledge base κ. It uses values from this knowledge base to deduce further labels until it is able to label all locations l in all reachable context pairs (S, θ_φ) either $\mathbf{t}$ or $\mathbf{f}$ with respect to φ. The knowledge base is a mapping from a context θ_φ and a location l to a ternary logic value.

Definition 3.3.6 (Knowledge base)**.** *Let $\mathcal{R}$ be an RKS. A* knowledge base $\kappa : L(\mathcal{R}) \times \Theta \to \mathbb{T}$ *for $\mathcal{R}$ is a function which satisfies*

$$\kappa(l, \langle \mathbf{m} \rangle \tau) = \kappa(l, \langle \mathbf{f} \rangle \tau) \bowtie \kappa(l, \langle \mathbf{t} \rangle \tau) \tag{3.2}$$

for all $l \in L(\mathcal{R})$ and all τ. The class of all knowledge bases is depicted by $\mathbb{K}$.

A lookup in the knowledge base $\kappa(l, \theta_\varphi)$ determines whether formula φ associated to the call context θ_φ holds in location l, when its sub-structure $S(l)$ is called in the context θ_φ. The knowledge base returns a ternary logic value. A value of $\mathbf{m}$ states that it is not known yet, whether the formula holds in the given context and location.

Equation 3.2 states that a value in an $\mathbf{m}$ context can be inferred by looking up the values in the $\mathbf{t}$ and $\mathbf{f}$ contexts and combining them with the $\bowtie$ operator. This is used by the algorithm, since it only checks in $\mathbf{t}$ and $\mathbf{f}$ contexts. The values in $\mathbf{m}$ contexts can then be inferred.

In the context of the algorithm, a location l in sub-structure $S(l)$ which is called in context θ_φ is said to be *labeled* with a formula φ, if the knowledge base indicates that the formula holds at that point ($\kappa(l, \theta_\varphi) = \mathbf{t}$).

It is important to define when a knowledge base is "complete" for a specific formula and what the correctness of a value in a knowledge base means:

Definition 3.3.7 (Fully and Correctly Labeled Knowledge Base)**.** *Let φ be a CTL formula and $\mathcal{R}$ an RKS. A knowledge base κ of $\mathcal{R}$ is said to be* fully labeled *with respect to φ, if all reachable configurations are either labeled $\mathbf{t}$ or $\mathbf{f}$:*

$$\begin{aligned} \forall \sigma \equiv \sigma' \oplus c \in \mathscr{S}(\mathcal{R}), l \in L(S_{\leftarrow c}) \,. \\ \kappa(l, \theta_\varphi(\sigma)) \in \{\mathbf{t}, \mathbf{f}\} \end{aligned} \tag{3.3}$$

κ is said to be correctly labeled *with respect to φ, if all values in it comply to the CTL semantics:*

$$\begin{aligned} \forall \sigma \equiv \sigma' \oplus c \in \mathscr{S}(\mathcal{R}), l \in L(S_{\leftarrow c}) \,. & \\ \kappa(l, \theta_\varphi(\sigma)) = \mathbf{t} \Rightarrow \mathcal{R}, (\sigma, l) \models \varphi & \\ \wedge\, \kappa(l, \theta_\varphi(\sigma)) = \mathbf{f} \Rightarrow \mathcal{R}, (\sigma, l) \not\models \varphi & \end{aligned} \tag{3.4}$$

A knowledge base is said to be fully and correctly labeled *with respect to φ, if it is fully labeled and correctly labeled with respect to φ. The following formula must hold in this case:*

$$\begin{aligned}\forall \sigma \equiv \sigma' \oplus c \in \mathscr{S}(\mathcal{R})\,, l \in L(S_{\leftarrow c})\,.& \\ \kappa(l, \theta_\varphi(\sigma)) = \mathbf{t} \Leftrightarrow \mathcal{R}, (\sigma, l) &\models \varphi \\ \wedge\, \kappa(l, \theta_\varphi(\sigma)) = \mathbf{f} \Leftrightarrow \mathcal{R}, (\sigma, l) &\not\models \varphi\end{aligned} \tag{3.5}$$

A knowledge base κ is said to be *fully labeled* with respect to a formula φ, if all locations in all reachable call contexts are either labeled $\mathbf{t}$ or $\mathbf{f}$, which means that there are no more undecided $\mathbf{m}$ values. A label is correct if it complies to the CTL semantics, that is, a label of $\mathbf{t}$ for a configuration (l, θ_φ) must imply that φ holds and $\mathbf{f}$ must imply that φ does not hold. A knowledge base is *correctly labeled* if all labels in it are correct. A *fully and correctly labeled* knowledge base in which each label is set correctly and all labels are actually decided to $\mathbf{t}$ or $\mathbf{f}$ is the goal of the labeling algorithm.

3.3.3. Obtaining Certain Call Contexts

Definition 3.3.2 on page 30 has shown how the call context of a specific call stack is defined. Although this definition is very important for the understanding of the call context semantics, it is not applicable for the labeling algorithm itself. The first problem with the definition is that it defines the call context over the validity of the formula to be checked (*valid*(φ, σ)). This semantic information is not known to the algorithm. The second problem of the definition is that it infers a call context for a call stack. As mentioned above, the algorithm explicitly avoids the usage of any call stack and instead uses only call contexts. Therefore, a call context of a call stack is not applicable for the algorithm. However, the algorithm does need to infer call contexts, because it must check the sub-structures under these contexts. Therefore, some functions are necessary which deduce call contexts from the limited information the algorithm has.

The algorithm needs two fundamental operations to gather reachable contexts: It must be possible to infer the initial call context under which the initial sub-structure is called (Informally: the call context for the entry point of the program). Next, given a call c in a sub-structure S and a call context under which S is called, it must be possible to infer the call context for c in this situation.

Given an RKS $\mathcal{R}$, the context assumption of the initial call stack $[c^{\text{in}}]$ depicts which formulae are valid in the virtual termination location l^{term} which is reached after the initial sub-structure has "returned". By definition, no atomic propositions hold in that location ($\mu(l^{\text{term}}) = \emptyset$) and the location is a trap location containing only a self-loop as transition. Using these characteristics of l^{term}, the validity of each CTL formula can be inferred. This is done by the initial call assumption function *ica* : CTL $\rightarrow \mathbb{T}$, which is defined recursively over the structure of φ:

$$\begin{aligned}
\mathit{ica}(\top) &= \mathbf{t} \\
\mathit{ica}(p) &= \mathbf{f} \\
\mathit{ica}(\neg\psi) &= \neg \mathit{ica}(\psi) \\
\mathit{ica}(\psi \vee \chi) &= \mathit{ica}(\psi) \vee \mathit{ica}(\chi) \\
\mathit{ica}(\mathbf{EG}\psi) &= \mathit{ica}(\psi) \\
\mathit{ica}(\mathbf{EX}\psi) &= \mathit{ica}(\psi) \\
\mathit{ica}(\mathbf{E}\psi\mathbf{U}\chi) &= \mathit{ica}(\chi)
\end{aligned}$$

The initial call context $icc : \mathrm{CTL} \to \Theta$ of a CTL a formula φ represents the call context of the initial call stack $[c^{\mathrm{in}}]$. Since this context should represent the validity of φ and its subformulae in the successor location of c^{in} (i.e., l^{term}), the *ica* function is used to deduce exactly these validity assertions. The *icc* function is defined recursively over the structure of φ:

$$\begin{array}{lcl}
icc(\top) & = & \langle ica(\varphi)\rangle\top \\
icc(p) & = & \langle ica(\varphi)\rangle p \\
icc(\neg\psi) & = & \langle ica(\varphi)\rangle(\neg icc(\psi)) \\
icc(\psi \vee \chi) & = & \langle ica(\varphi)\rangle(icc(\psi) \vee icc(\chi)) \\
icc(\mathbf{EG}\psi) & = & \langle ica(\varphi)\rangle\mathbf{EG}\; icc(\psi) \\
icc(\mathbf{EX}\psi) & = & \langle ica(\varphi)\rangle\mathbf{EX}\; icc(\psi) \\
icc(\mathbf{E}\psi\mathbf{U}\chi) & = & \langle ica(\varphi)\rangle\mathbf{E}\; icc(\psi)\mathbf{U}\; icc(\chi)
\end{array}$$

The function descends recursively into the structure of the formula and annotates each subformula with its initial context assumption *ica*.

As mentioned, the second operation which must be possible is to deduce the context of a call c in a sub-structure S which is called under another context θ'. This operation, written as $\theta(c, \theta')$, was depicted in Definition 3.3.5 on page 35. However, as stated there, the definition is non-operational since it needs to find a call stack having a specific call context and it uses the *valid* predicate, which requires perfect semantic validity information. This information is not known during the execution of the algorithm, since it is the goal of the algorithm to compute this very information. Therefore, the algorithm uses an operational pendant which looks up validity information from the knowledge base κ and inserts $\mathbf{m}$ assumptions when the validity is not known, yet. This pendant is the call context function *cc*. The $cc : \mathbb{K} \times C(\mathcal{R}) \times \Theta \to \Theta$ function takes a call c and a context θ under which the sub-structure $S(c)$ is called. The function returns the appropriate call context for call c. A call context of a specific call c should reflect which formulae hold in the successor location $l^{\mathrm{succ}}(c)$ of c under the given context θ. This information is found in the knowledge base κ. For all subformulae of the input context θ, the function looks up the validity of that subformula in the knowledge base κ and tags the resulting context accordingly. The function is defined recursively over the call context θ under which S was called:

$$\begin{array}{lcl}
\multicolumn{3}{l}{cc(\kappa, c, \theta) = \lambda(\theta) =} \\
\quad\lambda(\langle\mathbf{t}\rangle\top) & = & \langle\mathbf{t}\rangle\top \\
\quad\lambda(\langle\eta\rangle p) & = & \langle succVal\rangle p \\
\quad\lambda(\langle\eta\rangle\neg\theta_\psi) & = & \langle succVal\rangle\neg\lambda(\theta_\psi) \\
\quad\lambda(\langle\eta\rangle(\theta_\psi \vee \theta_\chi)) & = & \langle succVal\rangle(\lambda(\theta_\psi) \vee \lambda(\theta_\chi)) \\
\quad\lambda(\langle\eta\rangle\mathbf{EG}\theta_\psi) & = & \langle succVal\rangle\mathbf{EG}\lambda(\theta_\psi) \\
\quad\lambda(\langle\eta\rangle\mathbf{EX}\theta_\psi) & = & \langle succVal\rangle\mathbf{EX}\lambda(\theta_\psi) \\
\quad\lambda(\langle\eta\rangle\mathbf{E}\theta_\psi\mathbf{U}\theta_\chi) & = & \langle succVal\rangle\mathbf{E}\lambda(\theta_\psi)\mathbf{U}\lambda(\theta_\chi)
\end{array}$$

with $succVal = \kappa(l^{\mathrm{succ}}(c), \theta)$

The validity of the subformula in the successor location is denoted with *succVal*. It is looked up from the knowledge base κ and is inserted as context assumption in the resulting call context. To insert context assumptions for subformulae, the function descends recursively into the structure of the context. For notational conciseness, a curried version

$\lambda(\theta)$ of the cc function is used for the recursive calls. Since the parameters κ and c stay the same in each recursive call, they are curried in λ.

Note that cc only produces an accurate call context if κ is already sufficiently labeled with respect to the associated formula of the context θ and all its subformulae. Otherwise, the context may be "inaccurate", that is, it may contain **m** assumptions. This inaccuracy is properly handled by the algorithm.

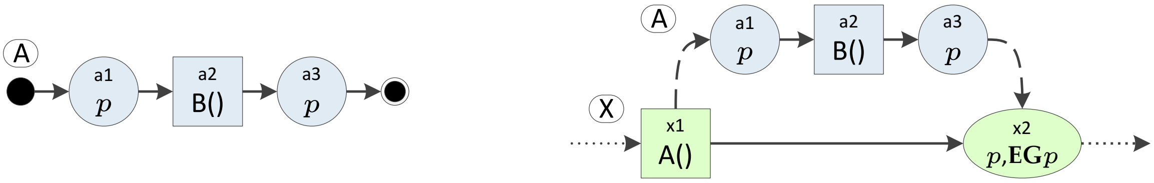


Figure 3.2.: An example sub-structure and its context

Example: The left side of Figure 3.2 shows a trivial sub-structure A which calls another sub-structure B. Assume κ to be an "oracle" knowledge base which is completely and correctly labeled with respect to all CTL formulae. A call to

$$cc(\kappa, \mathsf{a2}, \langle\mathbf{t}\rangle\mathbf{EG}\langle\mathbf{t}\rangle p)$$

would look up the call context of a2, if A is called in a context in which **EG**p and p hold. This situation is illustrated on the right side of Figure 3.2. The imaginary sub-structure X shows what the context $\langle\mathbf{t}\rangle\mathbf{EG}\langle\mathbf{t}\rangle p$ represents: It stands for a call x1 which calls A and has a successor location x2 in which p and **EG**p hold[3]. Given this context, the cc function determines the formulae which hold in the successor location of a2, which is a3. In this case, p would hold in a3, since this location has a p label. **EG**p would also hold, since the path from a3 to x2 exists which satisfies **G**p. Therefore, the call would yield the following result in this example:

$$cc(\kappa, \mathsf{a2}, \langle\mathbf{t}\rangle\mathbf{EG}\langle\mathbf{t}\rangle p) = \langle\mathbf{t}\rangle\mathbf{EG}\langle\mathbf{t}\rangle p$$

The input context influences the result, because the validity of the formula in the successor location of a call often depends on the validity of the formula in the context (i.e., in the successor location of the exit location, in this case pictured as imaginary location x2). For example, consider the result for the input context: $\langle\mathbf{f}\rangle\mathbf{EG}\langle\mathbf{t}\rangle p$:

$$cc(\kappa, \mathsf{a2}, \langle\mathbf{f}\rangle\mathbf{EG}\langle\mathbf{t}\rangle p) = \langle\mathbf{f}\rangle\mathbf{EG}\langle\mathbf{t}\rangle p$$

Now, **EG**p no longer holds in the context. Therefore, even though a3 and the context (x2) is labeled p, the negative context assumption for **EG**p ensures that **EG**p does not hold in the context. Therefore, **EG**p is also not satisfied in a3, because the only path from this location goes to the context, where it is known that no path satisfying **G**p exists, due to the negative context assumption. Thus, also the resulting context has **f** as context assumption for **EG**p.

Not only the context but also the rest of the sub-structure behind a call can affect the context of that call. For example, consider the same call for the context $\langle\mathbf{t}\rangle\mathbf{EG}\langle\mathbf{t}\rangle q$:

$$cc(\kappa, \mathsf{a2}, \langle\mathbf{t}\rangle\mathbf{EG}\langle\mathbf{t}\rangle q) = \langle\mathbf{f}\rangle\mathbf{EG}\langle\mathbf{f}\rangle q$$

[3] The label **EG**p shown location x2 in the figure denotes that the validity of **EG**p is assumed in this location. It is not an atomic proposition.

Now, the resulting context has both context assumptions set to **f**. Even if q and $\mathbf{EG}q$ hold in the context, neither of this formulae holds in a3, because a3 has no q label. Therefore, even though the formula holds in the context of A, it does not hold in the successor location of call a2.

Using the functions for the initial context *icc* and for the context of a specific call under another context *cc*, it is possible to find all reachable contexts and to determine which context must be used to check the sub-structure called by a certain call. This is used by the labeling algorithm which is presented in the next section.

3.4. The Labeling Algorithm

This section depicts the core algorithm for model checking recursive Kripke structures with CTL. Given an RKS $\mathcal{R}$, a formula φ which is to be checked and a knowledge base κ which is fully and correctly labeled with respect to all subformulae of φ (excluding φ itself, of course), the algorithm adds labels for φ to κ, yielding a fully and correctly labeled knowledge base with respect to φ. This is done by the *labelOneFormula* function depicted in Algorithm 3.1.

Algorithm 3.1 *labelOneFormula* : $\mathbb{K} \times \mathbb{R} \times \text{CTL} \to \mathbb{K}$

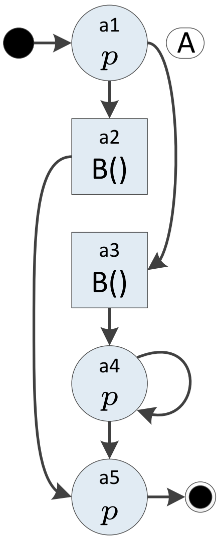
```
Precondition: $\kappa$ completely and correctly labeled with respect to all subformulae of $\varphi$.
fun labelOneFormula($\kappa, \mathcal{R}, \varphi$) =
  $\Xi \leftarrow collectContexts(\kappa, S_{in}(\mathcal{R}), icc(\varphi))$
  $\kappa \leftarrow computeLabels(\kappa, \Xi, \varphi)$
  return $\kappa$
```

As shown, the algorithm basically consists of two steps. The first step *collectContexts* collects all possibly reachable context pairs which are to be checked. The second step *computeLabels* then labels all locations in these contexts appropriately. The first step is shown in Algorithm 3.2.

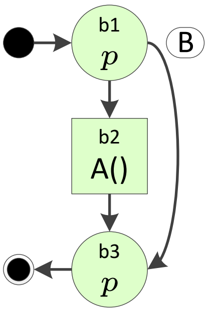
Algorithm 3.2 *collectContexts* : $\mathbb{K} \times \mathbb{S} \times \Theta \to \wp(\mathbb{S} \times \Theta)$

```
fun collectContexts($\kappa, S, \theta$) = $\lambda(\emptyset, \kappa, S, \theta)$
fun $\lambda(\Xi, \kappa, S, \theta \equiv \langle\eta\rangle\tau)$ =
  $\Xi \leftarrow \Xi \cup$ (if $\tau \equiv \top$ then $\{(S, \langle\mathbf{t}\rangle\tau)\}$ else $\{(S, \langle\mathbf{t}\rangle\tau), (S, \langle\mathbf{f}\rangle\tau)\}$)
  for all $c \in C(S)$ do
     $\langle\eta_c\rangle\tau_c \leftarrow cc(\kappa, c, \theta)$
     if $(S_{\leftarrow c}, \langle\mathbf{t}\rangle\tau_c) \notin \Xi$ then
        $\lambda(\Xi, \kappa, S_{\leftarrow c}, \langle\eta_c\rangle\tau_c)$
     end if
  end for
  return $\Xi$
```

The *collectContexts* function collects reachable context pairs which are to be labeled. The problem of collecting reachable context pairs in the call graph is simply accomplished by



Context	θ_1	θ_2	θ_3
	$\langle\mathbf{f}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{f}\rangle p$	$\langle\mathbf{f}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$	$\langle\mathbf{t}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$
State	call context / labels		
a1	$p, \mathbf{EG}p$	$p, \mathbf{EG}p$	$p, \mathbf{EG}p$
a2	$\langle\mathbf{f}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$	$\langle\mathbf{f}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$	$\langle\mathbf{t}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$
a3	$\langle\mathbf{t}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$	$\langle\mathbf{t}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$	$\langle\mathbf{t}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$
a4	$p, \mathbf{EG}p$	$p, \mathbf{EG}p$	$p, \mathbf{EG}p$
a5	p	p	$p, \mathbf{EG}p$



Context	θ_2	θ_3
	$\langle\mathbf{f}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$	$\langle\mathbf{t}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$
State	call context / labels	
b1	$p, \mathbf{EG}p$	$p, \mathbf{EG}p$
b2	$\langle\mathbf{f}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$	$\langle\mathbf{t}\rangle\mathbf{EG}$ $\vert$ $\langle\mathbf{t}\rangle p$
b3	p	$p, \mathbf{EG}p$

Figure 3.3.: The RKS $(\{A, B\}, A)$ and its associated contexts

a graph traversal which gathers traversed context pairs in the set Ξ. From the call context θ in which S is called, the call contexts for calls in S can be deduced using the *cc* function (line 3). These call contexts, together with the target sub-structure of the call yield the new context pairs to be traversed, but only if they are not visited yet (line 5 and 6). The extra function λ is used to carry the accumulator Ξ during the context collection.

Because the knowledge base does not yet contain information for the topmost formula of the context, the *cc* function cannot infer the correct context assumption for this formula, instead, it always inserts $\mathbf{m}$. Therefore, both context assumptions $\mathbf{t}$ and $\mathbf{f}$ have to be checked for the topmost subformula (line 1). An exception to this rule is the formula $\top$, which may only have $\mathbf{t}$ as context assumption.

The goal of line 5 is to skip the visiting of a call if its resulting context pair is already visited. Since the currently visited context is always inserted with $\mathbf{t}$ (and $\mathbf{f}$ if the formula is not $\top$) as topmost assumption, it is sufficient to check against $\langle\mathbf{t}\rangle\tau_c$ instead of $\langle\eta_c\rangle\tau_c$ to determine whether a call must be visited.

Example: Consider the RKS $(\{A, B\}, A)$ shown on the left side of Figure 3.3 and the formula

$\varphi = \mathbf{EG}p$. On the right side of the figure, tables depicting the call context of calls and labels of locations with respect to φ are displayed. The context in the first row of the tables depicts the context under which the sub-structure is called. The further rows depict the truth values for the subformulae of $\mathbf{EG}p$, that is $\mathbf{EG}p$ itself and p in the respective location. For calls, the rows instead depict the call context with respect to $\mathbf{EG}p$.

For example, the first column for location a5 depicts which of the subformulae of $\mathbf{EG}p$ are true in this location under the assumption that A is called in the context $\langle\mathbf{f}\rangle\mathbf{EG}\langle\mathbf{f}\rangle p$. In this case, the context states that $\mathbf{EG}p$ is not valid in the successor location of a5. Therefore, it also cannot be valid in a5. Because of this, only p is displayed for a5. In contrast, $\mathbf{EG}p$ and p are valid in location a4, since this location has a self-loop forming the infinite path where p holds. Because of that, the table shows p, $\mathbf{EG}p$ for this location to denote that both formulae hold.

As example for a call, consider the first column for a2 which depicts the call context of this call when sub-structure A is called under context $\langle\mathbf{f}\rangle\mathbf{EG}\langle\mathbf{f}\rangle p$. As a reminder, the call context is the formula φ annotated with the truth values in the successor location of the call. For call a2, the successor location is a5. As displayed in the row for this location, $\mathbf{EG}p$ does not hold there, but p holds. Therefore, the call context for a2 has $\mathbf{EG}p$ annotated with $\mathbf{f}$ and p with $\mathbf{t}$.

The table columns display all reachable contexts for the sub-structures with respect to formula $\varphi = \mathbf{EG}p$. The RKS "starts" in sub-structure A with the initial context $\langle\mathbf{f}\rangle\mathbf{EG}\langle\mathbf{f}\rangle p$ (first column for A). Here, the call a2 calls B with $\langle\mathbf{f}\rangle\mathbf{EG}\langle\mathbf{t}\rangle p$ (first column for B) and the call a3 calls B with $\langle\mathbf{t}\rangle\mathbf{EG}\langle\mathbf{t}\rangle p$ (second column B). In B, the call b2 calls A with context $\langle\mathbf{f}\rangle\mathbf{EG}\langle\mathbf{t}\rangle p$ (second column A) and $\langle\mathbf{t}\rangle\mathbf{EG}\langle\mathbf{t}\rangle p$ (third column A), depending on in which of the two reachable contexts B was called. No other contexts are reachable. Thus, the pairs $(A, \theta_1), (A, \theta_2), (A, \theta_3), (B, \theta_2), (B, \theta_3)$ depict all reachable equivalence classes for this formula.

For notational simplicity reasons, $\mathbf{EG}p$ contexts will be depicted by their two context assumptions in this example, ordered "top-down". For example, $\mathbf{tf}$ is used for the context $\langle\mathbf{t}\rangle\mathbf{EG}\langle\mathbf{f}\rangle p$. Thus, the reachable context pairs can be written as $(A, \mathbf{ff}), (A, \mathbf{ft}), (A, \mathbf{tt}), (B, \mathbf{ft}), (B, \mathbf{tt})$.

Now consider a knowledge base κ which is completely and correctly labeled with respect to $\mathbf{EG}p$. Thus, all labels displayed in the tables are also stored in κ. With this knowledge base, the *collectContexts* algorithm is called for the formula $\neg\mathbf{EG}p$. It is easy to deduce that the reachable contexts of $\neg\mathbf{EG}p$ are equal to the ones for $\mathbf{EG}p$ with the topmost context assumption being the opposite of the $\mathbf{EG}p$ context assumption (because, if $\mathbf{EG}p$ holds, then $\neg\mathbf{EG}p$ does not and vice versa). Again, a top-down notation for the context assumptions is used. For example, $\mathbf{ff}$ would be transformed to $\mathbf{tff}$ as follows:

$$\mathbf{ff} = \begin{array}{c} \langle\mathbf{f}\rangle\mathbf{EG} \\ | \\ \langle\mathbf{f}\rangle p \end{array} \qquad \Longrightarrow \qquad \mathbf{tff} = \begin{array}{c} \langle\mathbf{t}\rangle\neg \\ | \\ \langle\mathbf{f}\rangle\mathbf{EG} \\ | \\ \langle\mathbf{f}\rangle p \end{array}$$

Thus, the reachable context pairs for $\neg\mathbf{EG}p$ are exactly the pairs from $\mathbf{EG}p$ with "negated" contexts: $(A, \mathbf{tff}), (A, \mathbf{tft}), (A, \mathbf{ftt}), (B, \mathbf{tft}), (B, \mathbf{ftt})$. When executing the *collectContexts* algorithm for $\varphi = \neg\mathbf{EG}p$, these pairs should be collected into the set Ξ to yield a correct result. Of course, the reachable context pairs could be directly inferred from the reachable context pairs of ψ in the special case of $\varphi = \neg\psi$, as done above. This is not done by the algorithm for simplicity reasons. Instead, it performs a context exploration regardless of the formula to be checked.

The algorithm uses the *cc* function do deduce call contexts from the entries in the knowledge base. In case of $\varphi = \neg\mathbf{EG}p$, the knowledge base is completely and correctly labeled with respect to $\mathbf{EG}p$. However, it is not labeled at all with respect to $\neg\mathbf{EG}p$, yet. This implies that calls of *cc* will always yield the correct contexts, as displayed in the tables of Figure 3.3, with respect to $\mathbf{EG}p$. Since the assumption for $\neg\mathbf{EG}p$ is not yet present in the knowledge base yet, the *cc* function will always insert $\mathbf{m}$ as context assumption for this formula.

To exemplify how the context collection works, the *collectContexts* function is executed step by step for $\neg\mathbf{EG}p$. The example shows that the algorithm indeed collects all reachable context pairs:

1. The initial call to *collectContexts* is executed by function *labelOneFormula* with the initial sub-structure A and the initial context *icc*($\neg\mathbf{EG}p$), which is $\mathbf{tff}$. Because the topmost context assumption is often $\mathbf{m}$ in this step, the algorithm always inserts both contexts with $\mathbf{t}$ and $\mathbf{f}$ as topmost context assumption into Ξ. Thus, in the first step, the pairs $(A, \mathbf{tff})$ and $(A, \mathbf{fff})$ are inserted into Ξ[4]:

$$\begin{aligned} S &= A \\ \theta &= \mathbf{tff} \\ \Xi &= \{(A, \mathbf{tff}), (A, \mathbf{fff})\} \end{aligned}$$

2. The call $\mathsf{a2}$ is explored recursively. For this call, the call context *cc*(κ, $\mathsf{a2}$, $\mathbf{tff}$) is $\mathbf{mft}$. The bottom part $\mathbf{ft}$ can be looked up from the table in Figure 3.3 (row $\mathsf{a2}$, column θ_1). The upper $\mathbf{m}$ is the result of the missing information about $\neg\mathbf{EG}p$ in κ. In the recursive call, the context pairs $(B, \mathbf{tft}), (B, \mathbf{fft})$ are added:

$$\begin{aligned} S &= B \\ \theta &= \mathbf{mft} \\ \Xi &= \{(A, \mathbf{tff}), (A, \mathbf{fff}), (B, \mathbf{tft}), (B, \mathbf{fft})\} \end{aligned}$$

3. The call $\mathsf{b2}$ is explored, which has the context $\mathbf{mft}$. The call targets A and yields the new pairs $(A, \mathbf{tft}), (A, \mathbf{fft})$:

$$\begin{aligned} S &= A \\ \theta &= \mathbf{mft} \\ \Xi &= \begin{array}{l} \{(A, \mathbf{tff}), (A, \mathbf{fff}), (A, \mathbf{tft}), (A, \mathbf{fft}), \\ (B, \mathbf{tft}), (B, \mathbf{fft})\} \end{array} \end{aligned}$$

[4] Of course, the context $\mathbf{fff}$ is inconsistent, since an $\mathbf{f}$ assumption for $\mathbf{EG}p$ would imply a $\mathbf{t}$ assumption for $\neg\mathbf{EG}p$ yielding $\mathbf{tff}$. As already mentioned, inconsistent contexts pose no problem to the algorithm.

4. Under the current context $\mathbf{mft}$, the call a2 has the context $\mathbf{mft}$. The rewriting of the topmost assumption to true yields $\mathbf{tft}$ as context. Since the pair $(B, \mathbf{tft})$ is already in Ξ, a2 is not visited again.

   Next, the call a3 is checked. This one yields the not yet visited context $\mathbf{mtt}$. Thus, a3 is visited recursively and $(B, \mathbf{ttt}), (B, \mathbf{ftt})$ are added to Ξ:

$$\begin{aligned} S &= B \\ \theta &= \mathbf{mtt} \\ \Xi &= \begin{array}{l} \{(A, \mathbf{tff}), (A, \mathbf{fff}), (A, \mathbf{tft}), (A, \mathbf{fft}), \\ (B, \mathbf{ttt}), (B, \mathbf{ftt}), (B, \mathbf{tft}), (B, \mathbf{fft})\} \end{array} \end{aligned}$$

5. The call b2 has the context $\mathbf{mtt}$ under the current context $\mathbf{mtt}$. Since $(A, \mathbf{ttt})$ is not in Ξ yet, b2 is visited and $(A, \mathbf{ttt}), (A, \mathbf{ftt})$ are added to Ξ:

$$\begin{aligned} S &= A \\ \theta &= \mathbf{mtt} \\ \Xi &= \begin{array}{l} \{(A, \mathbf{tff}), (A, \mathbf{fff}), (A, \mathbf{tft}), (A, \mathbf{fft}), \\ (A, \mathbf{ttt}), (A, \mathbf{ftt}), \\ (B, \mathbf{ttt}), (B, \mathbf{ftt}), (B, \mathbf{tft}), (B, \mathbf{fft})\} \end{array} \end{aligned}$$

6. Both calls, a2 and a3, yield the context $\mathbf{mtt}$ under context $\mathbf{mtt}$. Since the context pair $(B, \mathbf{ttt})$ is already visited, the function returns to the previous step. Since no more unvisited calls are in B, the function returns again to step 3. Here, the second call a3 has to be checked. Under the context $\mathbf{mft}$, the call context of a3 is $\mathbf{mtt}$. This context is already visited, so the call is skipped and step 3 is finshed returning to step 2. Step 2 has no unvisited calls left and returns to step 1. Step 1 checks call a3 under context $\mathbf{tff}$, which yields $\mathbf{mtt}$ as call context for a3. This context is already visited, so the call is skipped and the function returns.

The execution trace of the context collection, which can also be used to infer the resulting set Ξ, can be described concisely by the sequence of visited context pairs. In this example, the sequence would be:

$$(A, \mathbf{tff}) \rightarrow (B, \mathbf{mft}) \rightarrow (A, \mathbf{mft}) \rightarrow (B, \mathbf{mtt}) \rightarrow (A, \mathbf{mtt})$$

The resulting set Ξ can be obtained from the sequence by adding two versions of each context pair in the sequence to Ξ: One with the topmost context assumption replaced by $\mathbf{t}$ and one with the assumption replaced by $\mathbf{f}$.

The set of reachable contexts for $\neg\mathbf{EG}p$ is $\{(A, \mathbf{tff}), (A, \mathbf{tft}), (A, \mathbf{ftt}), (B, \mathbf{tft}), (B, \mathbf{ftt})\}$. The set Ξ includes these contexts. However, it has twice as many context because each context was collected with $\mathbf{t}$ and $\mathbf{f}$ as topmost context assumption. The amount of contexts collected by the function is, in the worst case (which was shown here), twice as big as the number of reachable contexts.

It is very important for the correctness of the algorithm that *collectContexts* adds (at least) all reachable context pairs to the set Ξ. Therefore, this property is to be proven:

Lemma 3.4.1 (*collectContexts* collects all reachable contexts)**.** *Let $\mathcal{R}$ be an RKS and φ be a CTL formula. For each non-empty call stack $\sigma = \sigma' \oplus c \in \mathscr{S}(\mathcal{R})$, the corresponding context pair $(S_{\leftarrow c}, \theta_\varphi(\sigma'))$ with respect to φ is in Ξ as returned by collectContexts:*

$$\forall \sigma = \sigma' \oplus c \in \mathscr{S}(\mathcal{R}) \,.\, (S_{\leftarrow c}, \theta_\varphi(\sigma')) \in \Xi$$

Proof. By induction over the size of the call stack σ. The base case is the call stack $[c^{\text{in}}]$ with which the initial sub-structure is called. For this stack, the pair $(S^{\text{in}}, ic(\varphi))$ must be in Ξ. This pair contains exactly the parameters with which *collectContexts* is initially called by *labelOneFormula* (Algorithm 3.1, line 1). Therefore, this pair is surely collected.

For the induction step, it must be shown that the corresponding pair $(S_{\leftarrow c}, \theta_\varphi(\sigma))$ for a callstack $\sigma \oplus c = \sigma' \oplus c' \oplus c$ is in Ξ, assuming that the pair $(S_{\leftarrow c'}, \theta_\varphi(\sigma'))$ for σ' is included in Ξ:

Let $(S, \theta) = (S_{\leftarrow c}, \theta_\varphi(\sigma))$ and $(S', \theta') = (S_{\leftarrow c'}, \theta_\varphi(\sigma'))$. Furthermore let $\theta = \langle\eta\rangle\tau$ and $\theta' = \langle\eta'\rangle\tau'$. It has to be shown that $(S, \langle\eta\rangle\tau)$ is in Ξ. This is the case if λ was called for the pair $(S, \langle\eta_x\rangle\tau)$ with an arbitrary η_x. The context assumption η_x may be arbitrary, because the function inserts both possible contexts $\langle\mathbf{t}\rangle\tau$ and $\langle\mathbf{f}\rangle\tau$ into Ξ, irrespective of η_x (Algorithm 3.2, line 1). Thus, it has to be shown that λ is called for the pair $(S, \langle\eta_x\rangle\tau)$ for an arbitrary η_x.

Since $(S', \langle\eta'\rangle\tau') \in \Xi$ (induction hypothesis), *collectContexts* was called with $(S', \langle\eta_y\rangle\tau')$ (with arbitrary η_y). When looping over the calls of S', the call c' was picked in one iteration. In this iteration, the *cc* function was called with $cc(\kappa, c', \theta')$. Let $\theta^{cc} = \langle\eta^{cc}\rangle\tau^{cc}$ be the context returned by this call of *cc*. For all sub-contexts in τ^{cc}, *cc* returns the semantically correct call context, because of the precondition of *labelOneFormula*, which ensures that κ is completely and correctly labeled for all subformulae of φ. This semantically correct call context is exactly τ, so $\tau^{cc} = \tau$. For the topmost context assumption η^{cc}, *cc* always inserts $\mathbf{m}$ as context assumption, because κ has no truth value assigned for this context yet. Thus, the context returned by *cc* is $\langle\mathbf{m}\rangle\tau$ (Algorithm 3.2, line 3) . The sub-structure $S_{\leftarrow c'}$ which is called by c' is exactly S. Thus, line 5 now checks for the pair $(S, \langle\mathbf{t}\rangle\tau)$. If the pair is not yet included, λ is called with that pair. As depicted above, this implies that $(S, \langle\eta\rangle\tau)$ gets inserted into Ξ, what is to be shown. If $(S, \langle\eta\rangle\tau)$ is already in Ξ, then λ was already called earlier with $(S, \langle\eta_x\rangle\tau)$ for an arbitrary η_x. Therefore, $(S, \langle\eta\rangle\tau)$ is already inserted in this case, as well. □

An interesting aspect shown by the proof is that it is necessary to always collect both contexts with $\langle\mathbf{t}\rangle$ and $\langle\mathbf{f}\rangle$ as topmost context assumption, because it is not known yet in this phase which of these will be called by the sub-structure. This yields a small overhead, because there might be contexts which are not reachable but are checked nevertheless. This overhead is necessary, however.

After *collectContexts* has gathered all reachable context pairs, the labels for these contexts can be computed in the next step. This is done by the *computeLabels* function which is shown in Algorithm 3.3.

As shown in the algorithm, the *computeLabels* function does a case distinction over the structure of the formula φ for which labels should be generated and assigns labels according to the structure. The simple cases are handled in the function itself. The complex cases $\varphi \equiv \mathbf{EG}\psi$ and $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$ are handled by the *labelByFixpoint* function. The case $\varphi \equiv \mathbf{EX}\psi$

Algorithm 3.3 *computeLabels* : $\mathbb{K} \times \wp(\mathbb{S} \times \Theta) \times \mathrm{CTL} \to \mathbb{K}$

```
fun computeLabels($\kappa, \Xi, \varphi$) =
    if $\varphi \equiv \top$ then
        for all $(S, \theta_\varphi) \in \Xi, l \in L(S)$ do $\kappa(l, \theta_\varphi) \leftarrow \mathbf{t}$
    else if $\varphi \equiv p$ then
        for all $(S, \theta_\varphi) \in \Xi, l \in L(S)$ do $\kappa(l, \theta_\varphi) \leftarrow p \in \mu_S(l)$
    else if $\varphi \equiv \neg\psi$ then
        for all $(S, \theta_\varphi \equiv \langle\eta\rangle\neg\theta_\psi) \in \Xi, l \in L(S)$ do $\kappa(l, \theta_\varphi) \leftarrow \neg\kappa(l, \theta_\psi)$
    else if $\varphi \equiv \psi \vee \chi$ then
        for all $(S, \theta_\varphi \equiv \langle\eta\rangle(\theta_\psi \vee \theta_\chi)) \in \Xi, l \in L(S)$ do $\kappa(l, \theta_\varphi) \leftarrow \kappa(l, \theta_\psi) \vee \kappa(l, \theta_\chi)$
    else if $\varphi \equiv \mathbf{EX}\psi$ then
        for all $(S, \theta_\varphi \equiv \langle\eta\rangle\mathbf{EX}\theta_\psi \equiv \langle\eta\rangle\mathbf{EX}\langle\eta_\psi\rangle\tau_\psi) \in \Xi$ do
            for all $l \in L(S) \setminus \{l^{\mathrm{out}}(S)\}$ do
                $\kappa(l, \theta_\varphi) \leftarrow \beta(\kappa, c, \theta_\psi)$
            end for
            $\kappa(l^{\mathrm{out}}(S), \theta_\varphi) \leftarrow \eta_\psi$
        end for
    else if $\varphi \equiv \mathbf{EG}\psi \mid \mathbf{E}\psi\mathbf{U}\chi$ then
        $\kappa \leftarrow$ labelByFixpoint($\kappa, \Xi, \varphi$)
    end if
    return $\kappa$
```

uses the $\beta : \mathbb{K} \times C(\mathcal{R}) \times \Theta \to \mathbb{B}$ function which is defined as follows:

$$\beta(\kappa, c, \theta_\varphi) = \left(\bigvee_{l \in (\delta_{S(c)}(l) \cup L(S(c)))} \kappa(l, \theta_\varphi) \right) \vee \left(\bigvee_{c \in (\delta_{S(c)}(l) \cup C(S(c)))} \kappa(l^{\mathrm{in}}(S_{\leftarrow c}), cc(\kappa, c, \theta_\varphi)) \right) \tag{3.6}$$

For a call c in a sub-structure S which is called under context θ_φ, the β function "looks one state ahead": The function returns a boolean value stating whether any of the successors satisfies φ (which is exactly the semantics of $\mathbf{EX}\varphi$). For successor locations, the value is simply looked up in the knowledge base. For successor calls, the value is looked up in the knowledge base at the initial location of the called sub-structure $S_{\leftarrow c}$ under the resulting call context *cc*.

For compactness, the explanation of the *computeLabels* function is paired with a proof of its correctness:

Lemma 3.4.2 (*computeLabels* correctness)**.** *Let $\mathcal{R}$ be an RKS and φ a CTL formula. Let κ be a knowledge base which is correctly and completely labeled with respect to all subformulae of φ, according to Definition 3.3.7. Then, Algorithm 3.3 inserts only semantically correct values into κ with respect to φ. Thus, the call*

$$\kappa' \leftarrow \mathit{computeLabels}(\kappa, \Xi, \varphi)$$

yields a knowledge base κ' which is correctly labeled with respect to φ (cf. Equation 3.4 on page 36).

Equation 3.4 states that the labeling is correct for all reachable call stacks σ and all locations in the sub-structure $S_{\leftarrow\sigma}$ which is called by that call stack. This means that if a $\mathbf{t}$ or $\mathbf{f}$ label is found in the knowledge base for a location, then the configuration (σ, l) of the the location under the given call stack satisfies or does not satisfy the formula, respectively.

Proof. The lemma is proven by showing that each $\mathbf{t}$ or $\mathbf{f}$ value inserted into κ is semantically correct, which means that it must be possible at that moment to prove that the formula holds or does not hold, respectively. Here, it may be assumed that all values which are already contained in κ are correct, because the values for subformulae of φ were aleady inserted correctly as a precondition and newly inserted labels are always proven to be correct. For the case $\varphi \equiv \top$, all locations are labeled with $\mathbf{t}$ which is certainly the semantically correct labeling. For the case $\varphi \equiv p$, $\mathbf{t}$ is assigned if the label is present in the location and $\mathbf{f}$ otherwise. This is also trivially correct. The cases $\varphi \equiv \neg\psi$ and $\varphi \equiv \psi \vee \chi$ are similar: Here, the value of the subformula(e) are looked up in the knowledge base and the respective logic operation is performed. This is semantically correct, because it is assumed that the values for subformulae are certainly correctly inserted in the knowledge base.

In the case $\varphi \equiv \mathbf{EX}\psi$, all locations excluding exit locations l^{out} are labeled by checking if at least one successor satisfies ψ. This is done by the β function depicted in Equation 3.6 and already reflects the semantics of $\mathbf{EX}\psi$. Due to the precondition, the call context returned by *cc* is certainly the correct one, since the knowledge base is completely and correctly labeled for ψ. The labels for the exit locations l^{out} are directly deduced from the context assumption η_ψ (line 13), because this assumption reflects whether ψ holds in the successor state of l^{out}. Therefore, all locations are labeled semantically correct.

Finally, the cases $\varphi \equiv \mathbf{EG}\psi$ and $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$ are handled by the *labelByFixpoint* function. The correctness of this function is declared in Lemma 3.5.5 on page 55 and proven there.

□

3.5. Labeling EG and EU

The difficult cases are the path operators **EG** and **EU**. One of the most important contributions of this thesis is mainly the solution for formulae using these operators, as all other ones are rather trivial. In case of these operators, an incremental fixed point approach is used which computes some labels locally and then tries to deduce further labels from them. Once no more labels can be deduced from the previously computed ones, the fixed point is found and the remaining missing labels can be added without further checks. The *labelByFixpoint* function, as shown in Algorithm 3.4, tries to deduce further labels until no more labels can be deduced this way. The remaining $\mathbf{m}$ values are then resolved by the *decideRemainingMaybes* function.

The deduction of labels is done by performing local model checking of sub-structures. For this purpose, a usual so-called *associated Kripke structure* $\mathcal{K}$ is built from a single sub-structure. This non-recursive Kripke structure can then be checked by a usual CTL model checker for Kripke structures.

To construct the associated Kripke structure, the transition relation has to be made total. To achieve that, the exit state in the sub-structure must be given an outgoing transition.

Algorithm 3.4 *labelByFixpoint* : $\mathbb{K} \times \wp(\mathbb{S} \times \Theta) \times \text{CTL} \to \mathbb{K}$

```
fun labelByFixpoint($\kappa, \Xi, \varphi$) =
repeat
    $\kappa' \leftarrow \kappa$
    for all $(S, \theta_\varphi) \in \Xi$ do
        $\kappa \leftarrow$ deduceLabels($\kappa, S, \theta_\varphi$)
    end for
until $\kappa' = \kappa$    //Fixpoint found?
$\kappa \leftarrow$ decideRemainingMaybes($\kappa, \Xi, \varphi$)
return $\kappa$
```

In addition, the context assumption and the assumptions about the calls have to be represented in this Kripke structure. Definition 3.5.1 depicts the associated Kripke structure.

Definition 3.5.1 (Associated Kripke Structure). *Let $S = (L, l^{\text{in}}, l^{\text{out}}, C, \delta, \mu, \nu)$ be a sub-structure, κ be a knowledge base, and $\theta \equiv \langle\eta\rangle\mathbf{EG}\theta_\psi \mid \langle\eta\rangle\mathbf{E}\theta_\psi\mathbf{U}\theta_\chi$ be a call context. Let $AP = \{\ulcorner\psi\urcorner, \ulcorner\chi\urcorner, \ulcorner\varphi\urcorner, \ulcorner\varphi?\urcorner\}$ be a set of atomic propositions.*

Let $\alpha : \mathbb{T} \times AP \to \wp(AP)$ be a function which is defined as follows:

$$\alpha(t, p) = \begin{cases} \{p\} & \textit{if } t = \mathbf{t} \\ \emptyset & \textit{otherwise} \end{cases}$$

Let $\beta : \mathbb{T} \to \wp(AP)$ be a function which is defined as follows:

$$\beta(t) = \begin{cases} \{\ulcorner\varphi\urcorner\} & \textit{if } t = \mathbf{t} \\ \{\ulcorner\varphi?\urcorner\} & \textit{if } t = \mathbf{m} \\ \emptyset & \textit{otherwise} \end{cases}$$

The associated Kripke structure *$\mathcal{K}(\kappa, S, \theta) = (\mathcal{S}, I, \delta, \mu)$ is a Kripke structure over the set of atomic propositions AP with*

- *the set of states $\mathcal{S} = L \cup C \cup \{\omega\}$*
- *the set of initial states $I = \{l^{\text{in}}\}$*
- *the transition relation $\delta = \delta \cup \{(l^{\text{out}}, \omega), (\omega, \omega)\}$*
- *the labeling function $\mu : \mathcal{S} \to \wp(AP)$ which is defined as follows:*

$$\mu(l) = \begin{cases} \alpha(\kappa(l, \theta_\psi), \ulcorner\psi\urcorner) & \textit{if } \theta \equiv \langle\eta\rangle\mathbf{EG}\theta_\psi \\ \alpha(\kappa(l, \theta_\psi), \ulcorner\psi\urcorner) \cup \alpha(\kappa(l, \theta_\chi), \ulcorner\chi\urcorner) & \textit{if } \theta \equiv \langle\eta\rangle\mathbf{E}\theta_\psi\mathbf{U}\theta_\chi \end{cases} \qquad \textit{for all } l \in L$$

$$\mu(c) = \beta\left(\kappa\left(l^{\text{in}}(S_{\leftarrow c}), cc(\kappa, c, \theta)\right)\right) \qquad \textit{for all } c \in C$$

$$\mu(\omega) = \beta(\eta)$$

The sub-structure is transformed to a Kripke structure with a total transition relation by adding the state ω, which represents the successor location of the exit location l^{out} in the

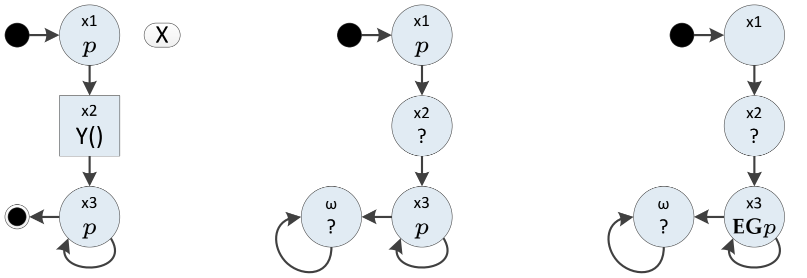


Figure 3.4.: A sub-structure (left) and its associate Kripke structures with respect to $\psi = p$ (middle) and $\psi = \mathbf{EG}p$ (right)

call context. A transition from l^{out} to ω is added to represent these semantics. In addition, the ω state receives a self-loop.

The most interesting aspect about the associated Kripke structure is that its atomic propositions are fixed and do not include the atomic propositions of the RKS. The propositions represent the validity of the formula φ and its subformulae ψ and χ: The propositions $\ulcorner\psi\urcorner$ and $\ulcorner\chi\urcorner$, which are only assigned to locations, state that the subformula ψ and χ, respectively, hold at the location. The absence of such label states that the respective formula does not hold at the location. The labels $\ulcorner\varphi\urcorner$ and $\ulcorner\varphi_?\urcorner$ represent statements about the validity of φ. These labels are only used in calls and in ω. Because φ is the formula which is to be resolved, it is not fully known yet, where φ holds. Therefore, two labels are necessary: The label $\ulcorner\varphi\urcorner$ represents that φ is known to hold and $\ulcorner\varphi_?\urcorner$ represents that it is unknown whether φ holds. No label at all means that it is known that φ does not hold in the respective call or in ω.

The labeling is performed as follows: Locations are labeled with $\ulcorner\psi\urcorner$ and $\ulcorner\chi\urcorner$ by looking up the validity of ψ and χ, respectively, in the knowledge base. Due to the precondition, the knowledge base is complete and correct for these formulae. The ω state is labeled accordingly to the context assumption η, because this value represents the assumption about the validity of φ in the successor location of the exit location l^{out}, which is exactly ω. Calls c are labeled according to the validity of φ in the initial location $l^{\text{in}}(S_{\leftarrow c})$ of the called sub-structure $S_{\leftarrow c}$. The call context under which this structure is called by c is inferred using the *cc* function.

Example: Consider the sub-structure X on the left side of Figure 3.4. In the middle and on the right, the figure depicts two associated Kripke structures $\mathcal{K}(\kappa, X, \theta)$ of X. As shown, an ω state has been added and calls are usual states in $\mathcal{K}$. In the Kripke structure in the middle, locations are labeled with $\psi = p$. This structure could be used for checking X against formulae which have p as direct sub-structure, like $\mathbf{EG}p$ or $\mathbf{E}p\mathbf{U}q$. On the right, locations are labeled with $\psi = \mathbf{EG}p$ (it is assumed that no path satisfying $\mathbf{G}p$ from location x1 exists). This associated Kripke structure could be used for formulae having $\mathbf{EG}p$ as subformula, like $\mathbf{EGEG}p$. The question marks in the omega state and in the call state mark that the labels in these states are dependent on the context and on information about the validity of the formula to be checked in the initial state of X. These labels will change during iterations of the fixed point algorithm, while the labels for the locations x1 and x3 will stay the same. Note that the actual labels in the associated Kripke structure are $\ulcorner\psi\urcorner$

and $\ulcorner\chi\urcorner$ instead of p or $\mathbf{EG}p$. The labels p and $\mathbf{EG}p$ were only used to symbolize that $\ulcorner\psi\urcorner$ in an associated Kripke structure always stands for the validity of a certain formula.

Using the associated Kripke structure, the function *deduceLabels*, as shown in Algorithm 3.5, tries to infer additional labels with respect to φ for a specific sub-structure S under a specific call context θ_φ.

Algorithm 3.5 *deduceLabels* : $\mathbb{K} \times \mathbb{S} \times \Theta \to \mathbb{K}$

```
fun deduceLabels($\kappa, S, \theta_\varphi \equiv \langle\eta\rangle\tau$) =
 1: $\varphi \leftarrow$ ( if $\tau \equiv \mathbf{EG}\theta_\psi$ then $\mathbf{EG}\ulcorner\psi\urcorner$ else $\mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\chi\urcorner$)
 2: for all $l \in L(S)$ do
 3:   if $\kappa(l, \theta_\varphi) = \mathbf{m}$ then
 4:     if $\mathcal{K}(\kappa, S, \theta_\varphi), l \models \varphi \vee \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner$ then
 5:       $\kappa(l, \theta_\varphi) \leftarrow \mathbf{t}$
 6:     else if $\mathcal{K}(\kappa, S, \theta_\varphi), l \models \neg(\varphi \vee \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner \vee \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi_?\urcorner)$ then
 7:       $\kappa(l, \theta_\varphi) \leftarrow \mathbf{f}$
 8:     end if
 9:   end if
10: end for
11: return $\kappa$
```

The function loops over all locations l which are not decided yet ($\kappa(l, \theta_\varphi) = \mathbf{m}$). Lines 4 and 6 use model checking of the associated Kripke structure to check whether a location can be marked $\mathbf{t}$ or $\mathbf{f}$.

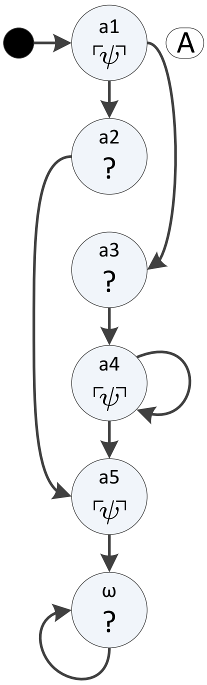
Example: Consider again the RKS $\mathcal{R} = (\{A, B\}, A)$ from Figure 3.3 on page 41. Figure 3.5 shows the associated Kripke structures for the sub-structures of $\mathcal{R}$ on the left and the steps of the deduce labels algorithm for $\varphi = \mathbf{EG}p$ on the right. The topmost table shows the column headers which are used for all tables below: Each column depicts the results for a sub-structure under a specific context. The first column for example depicts results for sub-structure A under context $\langle\mathbf{f}\rangle\mathbf{EG}\langle*\rangle p$. The asterisk depicts that the results are true for all contexts of that form regardless of the assumption about p. This is the case, because the context assumptions for p do not affect the result[5].

Per iteration of the loop within *labelByFixpoint*, there are two tables: The first one with caption "$\mathcal{K}$-Labeling" depicts the labels which the states of the associated Kripkes structure have. A label of $\ulcorner\varphi\urcorner$ in a call state depicts that it is assumed that $\mathbf{EG}p$ holds in this state. The assumptions in the ω states are directly taken from the context. For example, the context $\langle\mathbf{t}\rangle\mathbf{EG}\langle*\rangle p$ always results in a $\ulcorner\varphi\urcorner$ label in ω while an equal context with a false assumption for $\mathbf{EG}p$ results in no label. For locations, the label $\ulcorner\psi\urcorner$ was inserted where the location has a p label.

The second table with caption "κ-Update" depicts the content of the knowledge base κ after it was updated by the *deduceLabels* function in the iteration. For example, the value $\mathbf{m}$ for state a1 in the first column of the first κ-Update table depicts that $\kappa(\mathtt{a1}, \langle\mathbf{f}\rangle\mathbf{EG}\langle*\rangle p) = \mathbf{m}$ after the first iteration of the algorithm. At the beginning, all values in κ with respect to φ are $\mathbf{m}$. Values which change in an iteration are highlighted in green.

[5] Context assumptions for atomic propositions are only needed in the case of $\varphi = \mathbf{EX}p$

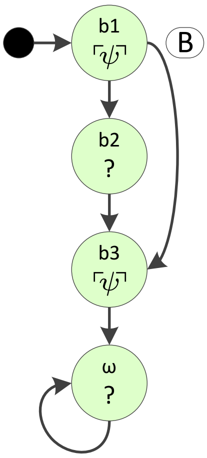
Sub	*A*			*B*	
Context	$\langle \mathbf{f} \rangle$**EG** | $\langle * \rangle p$	$\langle \mathbf{t} \rangle$**EG** | $\langle * \rangle p$		$\langle \mathbf{f} \rangle$**EG** | $\langle * \rangle p$	$\langle \mathbf{t} \rangle$**EG** | $\langle * \rangle p$



$\mathcal{K}$**-Labeling 1**					
a1	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$	b1	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$
a2	$\ulcorner\varphi_?\urcorner$	$\ulcorner\varphi_?\urcorner$	b2	$\ulcorner\varphi_?\urcorner$	$\ulcorner\varphi_?\urcorner$
a3	$\ulcorner\varphi_?\urcorner$	$\ulcorner\varphi_?\urcorner$	b3	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$
a4	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$	ω		$\ulcorner\varphi\urcorner$
a5	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$			
ω		$\ulcorner\varphi\urcorner$			

κ**-Update 1**					
a1	**m**	**m**	b1	**m**	**t**
a4	**t**	**t**	b3	**f**	**t**
a5	**f**	**t**			

$\mathcal{K}$**-Labeling 2**					
a1	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$	b1	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$
a2	$\ulcorner\varphi_?\urcorner$	$\ulcorner\varphi\urcorner$	b2	$\ulcorner\varphi_?\urcorner$	$\ulcorner\varphi_?\urcorner$
a3	$\ulcorner\varphi\urcorner$	$\ulcorner\varphi\urcorner$	b3	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$
a4	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$	ω		$\ulcorner\varphi\urcorner$
a5	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$			
ω		$\ulcorner\varphi\urcorner$			



κ**-Update 2**					
a1	**t**	**t**	b1	**m**	**t**
a4	**t**	**t**	b3	**f**	**t**
a5	**f**	**t**			

$\mathcal{K}$**-Labeling 3**					
a1	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$	b1	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$
a2	$\ulcorner\varphi_?\urcorner$	$\ulcorner\varphi\urcorner$	b2	$\ulcorner\varphi\urcorner$	$\ulcorner\varphi\urcorner$
a3	$\ulcorner\varphi\urcorner$	$\ulcorner\varphi\urcorner$	b3	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$
a4	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$	ω		$\ulcorner\varphi\urcorner$
a5	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$			
ω		$\ulcorner\varphi\urcorner$			

κ**-Update 3**					
a1	**t**	**t**	b1	**t**	**t**
a4	**t**	**t**	b3	**f**	**t**
a5	**f**	**t**			

Figure 3.5.: Example for the steps of the fixed point algorithm

For this example, the algorithm needs three iterations. In the first step, the associated Kripke structure $\mathcal{K}$ assumes $\ulcorner\varphi?\urcorner$ for each call, because no initial location of any sub-structure is already decided. After the first execution of *deduceLabels*, 7 values were decided. For example, the value for a4 changed to $\mathbf{t}$ in both contexts, because $\mathbf{EG}p$ always holds in a4 due to the p label and the self-loop in this state. The state b1 was labeled $\mathbf{t}$ in the context $\langle\mathbf{t}\rangle\mathbf{EG}\langle*\rangle p$, because the formula $\mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner$ holds in this state. In contrast, no result could be obtained for this state in the $\langle\mathbf{f}\rangle\mathbf{EG}\langle*\rangle p$ context, because the validity depends on the assumption about call b2 in this case: If the assumption turns out to be true, then $\mathbf{EG}p$ would hold in b1, otherwise it wouldn't.

In the second iteration, three assumptions about call a2 and a3 have been updated to assume that φ holds. This is the case, because $\mathbf{t}$ is now inserted as value for b1 in case of a $\langle\mathbf{t}\rangle\mathbf{EG}\langle*\rangle p$ context and the successor locations of these calls have a $\mathbf{t}$ value, thus yielding this very context. In case of call a2 in the first column, the value in κ for its successor location a5 is $\mathbf{f}$. For this call, the resulting context is $\langle\mathbf{f}\rangle\mathbf{EG}\langle*\rangle p$. For this context, $\mathbf{m}$ is still present in b1, so no assumption can be made for the call a2 in this case, yet. Due to the three updated assumptions, the values for a1 can be decided to be $\mathbf{t}$, because the path a1, a3 now satisfies $\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner$.

Because the second iteration has assigned $\mathbf{t}$ labels to the initial state a1 of A, the assumptions about call b2 are $\ulcorner\varphi\urcorner$ in iteration 3. This allows to decide the remaining $\mathbf{m}$ value in b1 to $\mathbf{t}$ due to the path b1, b2 satisfying $\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner$. Now, the knowledge base is completely labeled with respect to $\varphi = \mathbf{EG}p$. Thus, the final call to *decideRemainingMaybes* is unnecessary in this case.

For the correctness of the algorithm, it has to be shown that *deduceLabels* inserts only correct values into κ. Because the insertion of a value depends on the validity of a CTL formula in the associated Kripke structure, it has to be proven that the used CTL formulae are correct. The proof also shows why exactly these formulae were chosen.

Lemma 3.5.2 (*deduceLabels* correctness)**.** *Let $\mathcal{R}$ be an RKS and $\varphi \equiv \mathbf{EG}\psi \mid \mathbf{E}\psi\mathbf{U}\chi$ a CTL formula, both over atomic propositions AP. Let κ be a knowledge base for $\mathcal{R}$ which contains only correct entries and is completely labeled with respect to all subformulae of φ excluding φ itself. Let S be a sub-structure of $\mathcal{R}$ which is called under context θ_φ with respect to φ. Let φ' be a CTL formula over atomic propositions $AP' = \{\ulcorner\psi\urcorner, \ulcorner\chi\urcorner, \ulcorner\varphi\urcorner, \ulcorner\varphi?\urcorner\}$. The formula φ' is either $\mathbf{EG}\ulcorner\psi\urcorner$ if $\varphi \equiv \mathbf{EG}\psi$ or $\mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\chi\urcorner$ if $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$. The following formulae hold for all call stacks σ which have the call context $\theta_\varphi(\sigma) = \theta_\varphi$:*

$$\mathcal{K}(\kappa, S, \theta_\varphi), l \models \varphi' \vee \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner \Rightarrow \mathcal{K}(\mathcal{R}), (\sigma, l) \models \varphi \tag{3.7}$$

$$\mathcal{K}(\kappa, S, \theta_\varphi), l \models \neg(\varphi' \vee \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner \vee \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi?\urcorner) \Rightarrow \mathcal{K}(\mathcal{R}), (\sigma, l) \not\models \varphi \tag{3.8}$$

Proof. Let $\mathcal{K}^{\mathsf{ass}}$ depict the associated Kripke structure $\mathcal{K}(\kappa, S, \theta_\varphi)$. To show that the results obtained by model checking $\mathcal{K}^{\mathsf{ass}}$ comply to the semantics of RKSs which are defined over the semantic Kripke structure $\mathcal{K}(\mathcal{R})$, a mapping from the states of $\mathcal{K}^{\mathsf{ass}}$ to the ones of $\mathcal{K}(\mathcal{R})$ has to be found to argue which state in $\mathcal{K}^{\mathsf{ass}}$ can make assumptions about which state in $\mathcal{K}(\mathcal{R})$. A location $l \in L(S)$ in $\mathcal{K}^{\mathsf{ass}}$ is contained in $\mathcal{K}(\mathcal{R})$ as a single state. In these locations, a label of $\ulcorner\psi\urcorner$ or $\ulcorner\chi\urcorner$ in $\mathcal{K}^{\mathsf{ass}}$ is equal to the validity of ψ or χ, respectively, in $\mathcal{K}(\mathcal{R})$ at that location. The absence of such label signalizes that the respective formula is not valid in that location. The call states $c \in C(S)$ and the ω state require more thoughts. In $\mathcal{K}^{\mathsf{ass}}$, these are only simple states. In $\mathcal{K}(\mathcal{R})$, however, these are not single states but instead arbitrary "sub-graphs" of states. The only fact which is known about these sub-graphs is whether φ holds in their initial state, that is, the state which is equal to the initial location of the called sub-structure in case of call states or equal to the successor state of l^{out} in case of ω. A label of $\ulcorner\varphi\urcorner$ represents the validity of φ in that initial state, no label represents that φ is not valid in that state, and $\ulcorner\varphi?\urcorner$ represents that nothing is known about the validity of φ the initial state.

Equation 3.7: The algorithm labels a location $\mathbf{t}$ with respect to φ, if the formula $\xi = \varphi' \vee \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner$ is satisfied in the location in $\mathcal{K}^{\mathsf{ass}}$. For both possible formulae $\mathbf{EG}\psi$ and $\mathbf{E}\psi\mathbf{U}\chi$, a label of $\mathbf{t}$ means that φ holds and thus a path which is a witness for its validity exists. There are two cases for such path:

1. The path consists only of states which are locations in the sub-structure. In this case, the formula φ' is satisfied in $\mathcal{K}^{\mathsf{ass}}$ and so is ξ.
2. The path starts with locations of the sub-structure and then enters a call or ω. In this case, the path is a certain witness if all locations satisfy ψ (i.e., are labeled $\ulcorner\psi\urcorner$ in $\mathcal{K}^{\mathsf{ass}}$) and the formula is known to hold in the initial location of the state where the path "left" the sub-structure. This is exactly satisfied if $\mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner$ and thus ξ is satisfied in $\mathcal{K}^{\mathsf{ass}}$.

Because the formula φ holds if either of the paths exist, a location may be labeled $\mathbf{t}$ if the formula ξ is satisfied in $\mathcal{K}^{\mathsf{ass}}$.

Equation 3.8: The algorithm labels a location $\mathbf{f}$ with respect to φ, if the formula $\xi = \neg(\varphi' \vee \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner \vee \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi?\urcorner)$ is satisfied in $\mathcal{K}^{\mathsf{ass}}$. This formula is satisfied if φ' does not hold, $\mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner$ does not hold, and $\mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi?\urcorner$ does not hold. A label of $\mathbf{f}$ represents that φ is not valid and thus no witnessing path for φ exists. Now assume that the algorithm labels a location $\mathbf{f}$ although φ holds in it. Thus, a path witnessing path for φ exists. Again there are the two possibilities for such path:

1. The path consists only of states which are locations in the sub-structure. In this case, the formula φ' would be satisfied in $\mathcal{K}^{\mathsf{ass}}$. Consequently, ξ would not be satisfied and the algorithm would not label the location $\mathbf{f}$.
2. The path starts with locations of the sub-structure and then enters a call or ω. In this case, the path may be a witness if all locations satisfy ψ (and are thus labeled $\ulcorner\psi\urcorner$ in $\mathcal{K}^{\mathsf{ass}}$) and then the formula enters a sub-graph. For the assumption about the initial location of that sub-graph, three cases are possible:
   a) It is known that φ does not hold in this location (no label). Then the path cannot be a witness for φ.
   b) It is unknown whether φ holds in the location (label $\ulcorner\varphi?\urcorner$). Then, this path would also be a witness for the validity of $\mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi?\urcorner$ in $\mathcal{K}^{\mathsf{ass}}$. In this case, ξ would not hold and the algorithm would not label the location $\mathbf{f}$.
   c) φ is known to hold in the location (label $\ulcorner\varphi\urcorner$). Then, this path would also be a witness for the validity of $\mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner$ in $\mathcal{K}^{\mathsf{ass}}$. Again, ξ would not hold and the algorithm would not label the location $\mathbf{f}$.

Because all cases of paths witnessing the validity of φ lead to a contradiction, no such path can exist. Therefore, the label of $\mathbf{f}$ is correct. □

The *deduceLabels* function is executed by the *labelByFixpoint* function repeatedly until no more changes are made to the knowledge base. This must be done, because one execution of *deduceLabels* could label the initial location of a sub-structure. This leads to updated assumptions about calls calling that sub-structure and thus can yield new labels. However, it is often the case that no further changes are made but still some locations stay undecided. These locations are then labeled by the *decideRemainingMaybes* function depicted in Algorithm 3.6.

Algorithm 3.6 *decideRemainingMaybes* : $\mathbb{K} \times \wp(\mathbb{S} \times \Theta) \times \mathrm{CTL} \to \mathbb{K}$

```
fun decideRemainingMaybes($\kappa, \Xi, \varphi \equiv \mathbf{EG}\psi \mid \mathbf{E}\psi\mathbf{U}\chi$) =
for all $(S, \theta) \in \Xi, l \in L(S)$ do
  if $\kappa(l, \theta) = \mathbf{m}$ then
    if $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$ then
      $\kappa(l, \theta) \leftarrow \mathbf{f}$
    else if $\varphi \equiv \mathbf{EG}\psi$ then
      $\kappa(l, \theta) \leftarrow \mathbf{t}$
    end if
  end if
end for
return $\kappa$
```

This function is rather trivial. In the case of $\varphi \equiv \mathbf{EG}\psi$, all remaining $\mathbf{m}$ locations are labeled $\mathbf{t}$. In the case of $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$, all remaining $\mathbf{m}$ locations are labeled $\mathbf{f}$. The reason for this decision is depicted in the following proof of correctness.

Lemma 3.5.3 (Infinite $\mathbf{m}$ Path)**.** *If any* $\mathbf{m}$ *entries in the knowledge base* κ *exist after the fixed point has been found by Algorithm 3.4, they must be part of at least one infinite path in the semantic Kripke structure* $\mathcal{K}(\mathcal{R})$*. All values in* κ *are* $\mathbf{m}$ *for all configurations in that path.*

Proof. This lemma is proven by contradiction: Assume an $\mathbf{m}$ value is not part of an infinite path in $\mathcal{K}(\mathcal{R})$ in which all states have the value $\mathbf{m}$. Then, it is at least part of a finite path. Because the path is finite, it ends in a sub-structure. The last $\mathbf{m}$ in this path would only have successor states which are either labeled $\mathbf{t}$ or $\mathbf{f}$ (or no successor states at all). Therefore, the *updateGuarantees* function would have labeled this state either $\mathbf{t}$ or $\mathbf{f}$. Consequently, no such path can exist. □

Lemma 3.5.4 (*decideRemainingMaybes* correctness)**.** *The labeling performed by Algorithm 3.6 is correct.*

Proof. In the case of $\varphi \equiv \mathbf{EG}\psi$, the $\mathbf{m}$ values are labeled with $\mathbf{t}$. As Lemma 3.5.3 states, all $\mathbf{m}$ values must be part of an infinite path with $\mathbf{m}$ labels in each state. In this path, all locations must be labeled with ψ. This is the case, because otherwise the locations which are not labeled ψ would be labeled $\mathbf{f}$ by the algorithm, because $\mathbf{EG}\psi$ cannot hold in them. Since all locations in the infinite path are labeled ψ, the path represents a witness where

ψ always holds. This is equal to the semantics of $\mathbf{EG}\psi$, which thus holds. Therefore, all locations can be labeled $\mathbf{t}$.

In the case of $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$, the infinite path with $\mathbf{m}$ labeled locations may not contain a location which is labeled χ. If such location would exist in the path, it would have been decided to $\mathbf{t}$, since $\mathbf{E}\psi\mathbf{U}\chi$ trivially holds in this location. Since no locations with label χ exist in the path, it is no path where ψ holds until χ holds, which means it is no witness for the validity of φ.

All locations which are not in the set of $\mathbf{m}$ locations but are the target of a transition from a $\mathbf{m}$ labeled location must have the label $\mathbf{f}$; if one had a $\mathbf{t}$ label, the $\mathbf{m}$ location which has a transition to it would be labeled $\mathbf{t}$ by the algorithm, because $\mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner$ would hold in that location in the associated Kripke structure. Now assume there is a path satisfying $\psi\mathbf{U}\chi$ and starting at a $\mathbf{m}$ location. As shown above, this path cannot stay infinitely in the set of $\mathbf{m}$ locations. Therefore, the path must enter a non-$\mathbf{m}$ labeled location. As shown, this location must be labeled $\mathbf{f}$ which means that φ cannot hold in it. Therefore, that path cannot be a witness for the validity of φ.

Consequently, no path can be found which is a witness for the validity of $\mathbf{E}\psi\mathbf{U}\chi$ and therefore the $\mathbf{f}$ label is correct. □

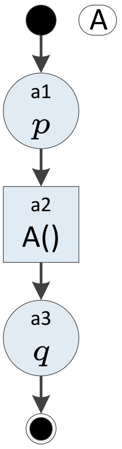
The *decideRemainingMaybes* function is the last part of the labeling algorithm for $\varphi \equiv \mathbf{EG}\psi$ and $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$. It resolves all remaining $\mathbf{m}$ locations and therefore yields a knowledge base which is completely and correctly labeled with respect to φ. Note that the function is only necessary in case of (possibly mutually) recursive calls. Otherwise, the fixed point mechanism is always able to infer all labels. However, even in the case of recursion, it is not always necessary, as shown in the previous example which contained mutually recursive sub-structures but did resolve all $\mathbf{m}$ values by fixed point iteration.

Example: Consider the RKS $\mathcal{R} = (\{A\}, A)$ over propositions $AP = \{p, q\}$ depicted on the left side of Figure 3.6. The only sub-structure A calls itself in call $\mathsf{a2}$, yielding an infinite looping recursion. The table on the right of the figure shows the execution of the fixed point algorithm for the formulae $\mathbf{EG}p$ and $\mathbf{E}p\mathbf{U}q$. In the first step, the algorithm is able to assign labels to $\mathsf{a3}$. However, the algorithm already terminates after this step, because it cannot update any assumptions about $\mathsf{a2}$, since these would depend on the labels in $\mathsf{a1}$. For these formulae, the *decideRemainingMaybes* would be used to label $\mathsf{a1}$. For $\varphi = \mathbf{EG}p$, the state would be labeled $\mathbf{t}$. This is correct, since there is an infinite path satisfying p in each state: the looping recursion path $([c^{\mathrm{in}}], \mathsf{a1}), ([c^{\mathrm{in}}, \mathsf{a2}], \mathsf{a1}), ([c^{\mathrm{in}}, \mathsf{a2}, \mathsf{a2}], \mathsf{a1}), \ldots$.

For the case of $\varphi = \mathbf{E}p\mathbf{U}q$, the location $\mathsf{a1}$ is labeled $\mathbf{f}$. This is also correct, because no path exists which eventually reaches $\mathsf{a3}$ containing the q label, because the looping recursion never terminates.

Lemma 3.5.5 (*labelByFixpoint* is correct)**.** *All values added to the knowledge base during the execution of the labelByFixpoint function are semantically correct.*

Proof. During the execution of *labelByFixpoint*, values are added to the knowledge base by the function *deduceLabels* and *decideRemaingMaybes*. Lemma 3.5.2 states the correctness of the former one, Lemma 3.5.4 states the correctness of the latter one. □



Context	$\langle\mathbf{f}\rangle\mathbf{EG}\langle*\rangle p$	$\langle\mathbf{t}\rangle\mathbf{EG}\langle*\rangle p$	$\langle\mathbf{f}\rangle\mathbf{E}\langle*\rangle p\mathbf{U}\langle*\rangle q$	$\langle\mathbf{t}\rangle\mathbf{E}\langle*\rangle p\mathbf{U}\langle*\rangle q$
$\mathcal{K}$-Labeling 1				
a1	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$	$\ulcorner\psi\urcorner$
a2	$\ulcorner\varphi?\urcorner$	$\ulcorner\varphi?\urcorner$	$\ulcorner\varphi?\urcorner$	$\ulcorner\varphi?\urcorner$
a3			$\ulcorner\chi\urcorner$	$\ulcorner\chi\urcorner$
a$^\omega$		$\ulcorner\varphi\urcorner$		$\ulcorner\varphi\urcorner$
κ-Update 1				
a1	**m**	**m**	**m**	**m**
a3	**f**	**f**	**t**	**t**

Figure 3.6.: The fixed point algorithm yielding remaining **m** values

Proposition 3.5.6 (Overall correctness and completeness)**.** *Let φ be a CTL formula, $\mathcal{R}$ an RKS, and κ a knowledge base which is correctly and completely labeled with respect to all subformulae of φ (excluding φ itself). Then, the call*

$$\kappa' \leftarrow \mathit{labelOneFormula}(\kappa, \mathcal{R}, \varphi)$$

returns a knowledge base κ' which is correctly and completely labeled with respect to φ.

Proof. It has to be proven that Equation 3.5 holds for φ, given that it holds for all subformulae of φ. Here is equation 3.5 again:

$$\begin{aligned} &\forall l \in L(S_{\leftarrow c})\,.\\ &\quad \kappa'(l, \theta_\varphi(\sigma)) = \mathbf{t} \Leftrightarrow \mathcal{R}, (\sigma, l) \models \varphi\\ &\wedge \kappa'(l, \theta_\varphi(\sigma)) = \mathbf{f} \Leftrightarrow \mathcal{R}, (\sigma, l) \not\models \varphi \end{aligned}$$

The equivalences can be split into two implications yielding the following two formulae to be proven:

$$\begin{aligned} &\forall l \in L(S_{\leftarrow c})\,.\\ &\quad \kappa'(l, \theta_\varphi(\sigma)) = \mathbf{t} \Rightarrow \mathcal{R}, (\sigma, l) \models \varphi\\ &\wedge \kappa'(l, \theta_\varphi(\sigma)) = \mathbf{f} \Rightarrow \mathcal{R}, (\sigma, l) \not\models \varphi \end{aligned} \tag{3.9}$$

and

$$\begin{aligned} &\forall l \in L(S_{\leftarrow c})\,.\\ &\quad \kappa'(l, \theta_\varphi(\sigma)) = \mathbf{t} \Leftarrow \mathcal{R}, (\sigma, l) \models \varphi\\ &\wedge \kappa'(l, \theta_\varphi(\sigma)) = \mathbf{f} \Leftarrow \mathcal{R}, (\sigma, l) \not\models \varphi \end{aligned} \tag{3.10}$$

Equation 3.9 is exactly the definition of a correctly labeled knowledge base. Lemma 3.4.2 states that the algorithm achieves this.

Equation 3.10 is satisfied if all reachable configurations can be looked up correctly in the knowledge base. Since Equation 3.9 already states that all labels are correct, the only thing which is missing to prove Equation 3.10 is to show that all labels are actually either labeled $\mathbf{t}$ or $\mathbf{f}$. This is exactly the definition of a fully labeled knowledge base:

$$\begin{aligned} \forall \sigma \equiv \sigma' \oplus c \in \mathscr{S}(\mathcal{R})\,, l \in L(S_{\leftarrow c})\,. \\ \kappa(l, \theta_\varphi(\sigma)) \in \{\mathbf{t}, \mathbf{f}\} \end{aligned} \tag{3.11}$$

Lemma 3.4.1 states that *collectContexts* collects all reachable context pairs. As depicted by the lemma, these pairs represent the contexts of all reachable call stacks. Thus, it suffices to show that all locations in these contexts are labeled either $\mathbf{t}$ or $\mathbf{f}$ by the algorithm.

It is easy to show that *computeLabels* labels all locations in all the collected contexts: For all cases except **EG** and **EU** the algorithm loops over all contexts and all locations in them and labels all of these locations. For the remaining cases, the *decideRemainingMaybes* function labels all previously unlabeled locations in all contexts. Therefore, all locations are labeled either $\mathbf{t}$ or $\mathbf{f}$ after the execution of *computeLabels*. Since the resulting knowledge base of *computeLabels* is used as result for *labelOneFormula*, the resulting knowledge base κ' is correctly and completely labeled with respect to φ. □

3.6. Model Checking Using the Labeling Algorithm

To use the labeling algorithm for model checking of a formula φ, a knowledge base must be built which is completely and correctly labeled with respect to φ. Usually, the user is only interested in the validity of φ in the initial state, which means the initial location of the initial sub-structure under the initial call context. This value can be looked up in the labeled knowledge base.

The labeling algorithm *labelOneFormula* performs the labeling with respect to one formula, given that all subformulae are already labeled in the input knowledge base. To use this algorithm for model checking of φ, it must be called recursively for subformulae of φ to yield a knowledge base which can be used as input for the checking of φ. This is done by the function *label* depicted in Algorithm 3.7.

Algorithm 3.7 *label* : $\mathbb{K} \times \mathbb{R} \times \mathrm{CTL} \to \mathbb{K}$

```
fun label($\kappa, \mathcal{R}, \varphi$) =
    if $\varphi \equiv \neg\psi \mid \mathbf{EX}\psi \mid \mathbf{EG}\psi$ then    //Recursive descent
        $\kappa \leftarrow$ label($\kappa, \mathcal{R}, \psi$)
    else if $\varphi \equiv \psi \vee \chi \mid \mathbf{E}\psi\mathbf{U}\chi$ then
        $\kappa \leftarrow$ label($\kappa, \mathcal{R}, \psi$)
        $\kappa \leftarrow$ label($\kappa, \mathcal{R}, \chi$)
    end if
    $\kappa \leftarrow$ labelOneFormula($\kappa, \mathcal{R}, \varphi$)    //Label $\varphi$
    return $\kappa$
```

The function takes a knowledge base κ, an RKS $\mathcal{R}$, and a formula φ to be checked and returns a knowledge base which is completely and correctly labeled with respect to φ. First, a recursive call of the function for subformulae of φ is performed to retrieve a knowledge base which is labeled with respect to these subformulae. Afterwards, the *labelOneFormula* function is used to label κ with respect to φ itself.

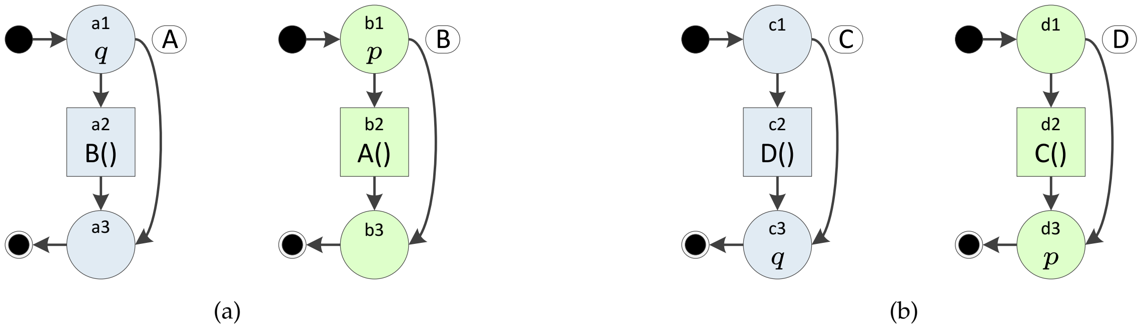


Figure 3.7.: Two recursive Kripke structures

The *modelcheck* function, as depicted by Algorithm 3.8 uses the *label* function to perform local model checking of a formula. Given an RKS $\mathcal{R} \equiv (\mathbb{S}, S^{\text{in}})$ and a formula φ, *modelcheck* returns whether φ is valid in the initial configuration $([c^{\text{in}}], l^{\text{in}}(S^{\text{in}}))$ of $\mathcal{R}$.

Algorithm 3.8 *modelcheck* : $\mathbb{R} \times \text{CTL} \to \mathbb{B}$

```
fun modelcheck($\mathcal{R} \equiv (\mathbb{S}, S^{\text{in}}), \varphi$) =
    $\kappa \leftarrow (\_ \mapsto \mathbf{m})$    //Init $\kappa$ with “no knowledge yet”
    $\kappa \leftarrow label(\kappa, \mathcal{R}, \varphi)$    //Perform labeling
    return $\kappa(l^{\text{in}}(S^{\text{in}}), ic(\varphi))$    //Lookup value in initial location
```

Theorem 3.6.1 (*modelcheck* correctness)**.** *Given an RKS $\mathcal{R} = (\mathbb{S}, S^{\text{in}})$ and a CTL formula φ, the result of modelcheck$(\mathcal{R}, \varphi)$ complies to the CTL semantics:*

$$\textit{modelcheck}(\mathcal{R}, \varphi) \Leftrightarrow \mathcal{R}, ([c^{\text{in}}], l^{\text{in}}(S^{\text{in}})) \models \varphi$$

Proof. Proposition 3.5.6 states that *labelOneFormula* produces a knowledge base κ' which is completely and correctly labeled with respect to φ, given that the input knowledge base κ is labeled completely and correctly with respect to all subformulae of φ. The *label* function performs the recursive labeling for subformulae of φ before φ itself is labeled (bottom-up). Therefore, *label* yields a correctly and completely labeled knowledge base.

The value for the initial configuration $([c^{\text{in}}], l^{\text{in}}(S^{\text{in}}))$ is looked up in the knowledge base using the initial location of the initial sub-structure under the initial context $ic(\varphi)$ with respect to φ. Since the knowledge base is completely and correctly labeled, this look-up surely yields a correct result. □

The examples presented up to this page only contain model checking of shallow formulae. To show how a nested formula is checked by the algorithm, a comprehensive example with a deeply nested formula is to be shown.

Example: Figure 3.7 shows two similar mutually recursive RKSs over the atomic propositions $AP = \{p, q\}$. While the one in Figure 3.7(a) has the initial states labeled, the one in Figure 3.7(b) has the exit states labeled. Consider the CTL formula $\varphi = \mathbf{EF}(p \wedge \mathbf{EF}(q \wedge \mathbf{EF}(p \wedge \mathbf{EF}(q \wedge \mathbf{EF}p))))$. The formula describes that there must be a path which contains p,. . . ,q,. . . ,p,. . . ,q,. . . ,p. It is easy to see that the formula holds in both RKSs; the satisfying path is the one that descends at least 5 times into the recursion. Although such formula might not be used in real applications, it is useful for showing how the algorithm is able to "look far enough" into the recursion to solve a formula. To model check the formula with the algorithm, it first has to be transformed into an equivalent formula using only the orthogonal operators. Because the transformation of $\psi \wedge \chi$ to $\neg(\neg\psi \vee \neg\chi)$ would introduce too many unimportant negation steps, consider that the algorithm is able to model check $\wedge$ explicitly without transformation. This can be done (and is done indeed in the implementation) by adding a $\psi \wedge \chi$ case to Algorithm 3.3 on page 46. This case is equal to the $\psi \vee \chi$ case but uses $\wedge$ instead of $\vee$ to combine the subformula values from the knowledge base. The operator **EF**ψ is transformed to $\mathbf{E}\top\mathbf{U}\psi$, yielding the following representation:

$$\varphi = \mathbf{E}\top\mathbf{U}(p \wedge \mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}(p \wedge \mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}p))))$$

The labeling starts with the innermost formulae which are $\top$ and p. The algorithm trivially labels all contexts with these propositions. The first interesting case is the innermost temporal operator $\mathbf{E}\top\mathbf{U}p$. In order to compactly depict which context pairs are collected, a short notation is used again: Since the formula $\top$ has always the context assumption $\mathbf{t}$ and the context assumption for atomic propositions like p is not important when not checking **EX** formulae, these assumptions are omitted. The only assumptions which are written are the ones of the temporal **EU** formula. To shorten the representation even more, only the context assumptions are shown, not the formula. For example, the context $\langle\mathbf{f}\rangle\mathbf{E}\langle*\rangle\top\mathbf{U}\langle*\rangle p$ is denoted by $\mathbf{f}$ and the context $\langle\mathbf{t}\rangle\mathbf{E}\langle*\rangle\top\mathbf{U}(\langle*\rangle q \wedge \langle\mathbf{f}\rangle\mathbf{E}\langle*\rangle\top\mathbf{U}\langle*\rangle p)$ by $\mathbf{tf}$.

At first, consider the RKS $\mathcal{R}_a = (\{A, B\}, A)$ from Figure 3.7(a). For the formula $\varphi = \mathbf{E}\top\mathbf{U}p$, the context pairs $(A, \mathbf{f}), (A, \mathbf{t}), (B, \mathbf{f}), (B, \mathbf{t})$ are collected. The results of the labeling algorithm for these context pairs are shown in Table 3.1.

Table 3.1.: Validity of $\mathbf{E}\top\mathbf{U}p$ in $\mathcal{R}_a$

Context	f	t
a1	$\mathbf{t}(2)$	$\mathbf{t}(1)$
a3	$\mathbf{f}(1)$	$\mathbf{t}(1)$
b1	$\mathbf{t}(1)$	$\mathbf{t}(1)$
b3	$\mathbf{f}(1)$	$\mathbf{t}(1)$

The numbers behind the validity values show in which iteration of the algorithm the result was obtained. Results in iteration x are usually based on results obtained in iteration $x - 1$. In case of a $\mathbf{t}$ context, the formula always holds because the path to the exit state satisfies the formula $\top\mathbf{U}\varphi$. In the $\mathbf{f}$ context, the formula holds for the initial state b1, since it is labeled with p. Consequently, it also holds for a1 due to the path a1a2 which calls B and ultimately reaches the p label there. The value for a1 is the only one which needs a call assumption and therefore can only be calculated in the second iteration of the algorithm[6].

The next formula to be checked is $\varphi = \mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}p)$. The collection starts with the context pair $(A, \mathbf{ff})$ which also collects the pair $(A, \mathbf{tf})$. Thus, the first collect step looks like

[6] Note that it might be the case that it could be labeled in the first iteration, if sub-structure B is computed before A. Such order-dependent effects are left out from the example and the iteration number in the worst case is shown.

this:

$$\begin{aligned}S &= A\\ \theta &= \mathbf{ff}\\ \Xi &= \{(A, \mathbf{ff}), (A, \mathbf{tf})\}\end{aligned}$$

The call a2 has the call context $\mathbf{mf}$ under the context $\mathbf{ff}$, thus the pair $(A, \mathbf{mf})$ is visited:

$$\begin{aligned}S &= B\\ \theta &= \mathbf{mf}\\ \Xi &= \{(B, \mathbf{ff}), (B, \mathbf{tf})(A, \mathbf{ff}), (A, \mathbf{tf})\}\end{aligned}$$

Under context $\mathbf{mf}$, the call b2 has the context $\mathbf{mf}$, which is already visited[7], so the collection stops. Note that the number of context pairs has not increased. The result of the labeling algorithm is shown in Table 3.2.

Table 3.2.: Validity of $\mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}p)$ in $\mathcal{R}_a$

Context	ff	tf
a1	$\mathbf{t}(2)$	$\mathbf{t}(1)$
a3	$\mathbf{f}(1)$	$\mathbf{t}(1)$
b1	$\mathbf{t}(1)$	$\mathbf{t}(1)$
b3	$\mathbf{f}(1)$	$\mathbf{t}(1)$

The result is the same as for the formula $\mathbf{E}\top\mathbf{U}p$: For the $\mathbf{tf}$ context, the formula holds in each state since the path going to the exit state satisfies the formula $\top\mathbf{U}\varphi$. For the $\mathbf{ff}$ context, the formula is satisfied in a1, because q and the subformula $\mathbf{E}\top\mathbf{U}p$ hold there. Consequently, it also holds in b1 due to the call b2 calling A.

It is easy to conclude that the number of contexts, when checking the remaining formulae, will not increase, and the validity of these formulae will all be equal. Consequently, the initial location will be labeled $\mathbf{t}$ for all formulae and the algorithm will return that the formula $\mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}p)$ thus holds.

In conclusion, the algorithm is able to "look deep enough" into the recursion without having to increase the number of contexts when checking the $\mathcal{R}_a$. The case is different when checking the RKS $\mathcal{R}_b = (\{C, D\}, C)$ depicted in Figure 3.7(b). Since the p and q labels are now behind the calls, the algorithm must infer the validity in the mutual recursion from the call context. For the first subformula $\varphi = \mathbf{E}\top\mathbf{U}p$, the context pairs which are collected are $(C, \mathbf{f}), (C, \mathbf{t}), (D, \mathbf{f}), (D, \mathbf{t})$. The resulting labeling is depicted in Table 3.3.

Again, the labels in the $\mathbf{t}$ context are all $\mathbf{t}$ due to the path which leaves the exit state and reaches the context satisfying $\top\mathbf{U}\varphi$. For the $\mathbf{f}$ context, the formula is true in d1 and d3, since d3 is labeled with p. Finally, the formula also holds in c1, because the call to D which reaches d1.

[7] More precisely, the corresponding $\mathbf{t}$ context $\mathbf{tf}$ is in Ξ, so the call is not explored.

Table 3.3.: Validity of $\mathbf{E}\top\mathbf{U}p$ in $\mathcal{R}_b$

Context	f	t
c1	$\mathbf{t}(2)$	$\mathbf{t}(1)$
c3	$\mathbf{f}(1)$	$\mathbf{t}(1)$
d1	$\mathbf{t}(1)$	$\mathbf{t}(1)$
d3	$\mathbf{t}(1)$	$\mathbf{t}(1)$

The next formula to be checked is $\varphi = \mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}p)$. The context collection starts as follows for this formula:

$$\begin{aligned} S &= C \\ \theta &= \mathbf{ff} \\ \Xi &= \{(C, \mathbf{ff}), (C, \mathbf{tf})\} \end{aligned}$$

The call context for call c2 is $\mathbf{mf}$ in this case, so D is visited under this context:

$$\begin{aligned} S &= D \\ \theta &= \mathbf{mf} \\ \Xi &= \{(D, \mathbf{ff}), (D, \mathbf{tf}), (C, \mathbf{ff}), (C, \mathbf{tf})\} \end{aligned}$$

This is the point where things start to get different from the previous example. In the case of a $\mathbf{mf}$ context, the subformula $\mathbf{E}\top\mathbf{U}p$ is valid in d3, therefore, the resulting call context for d2 is $\mathbf{mt}$, which is not visited yet, leading to the following next step:

$$\begin{aligned} S &= C \\ \theta &= \mathbf{mt} \\ \Xi &= \begin{array}{l} \{(C, \mathbf{ft}), (C, \mathbf{tt}), (D, \mathbf{ff}), \\ (D, \mathbf{tf}), (C, \mathbf{ff}), (C, \mathbf{tf})\} \end{array} \end{aligned}$$

Under this context, also call c2 has a $\mathbf{mt}$ context, so the final step also collects the two remaining context pairs:

$$\begin{aligned} S &= D \\ \theta &= \mathbf{mt} \\ \Xi &= \begin{array}{l} \{(D, \mathbf{ft}), (D, \mathbf{tt}), (C, \mathbf{ft}), (C, \mathbf{tt}), \\ (D, \mathbf{ff}), (D, \mathbf{tf}), (C, \mathbf{ff}), (C, \mathbf{tf})\} \end{array} \end{aligned}$$

In contrast to the previous example, all possible contexts where collected here. This leads to the labeling depicted in Table 3.4.

The contexts $\mathbf{tf}$ and $\mathbf{tt}$, which are depicted on the right, are the easiest: Since the formula holds in the context, the path reaching the exit state surely satisfies the formula. For the other cases, d3 is labeled $\mathbf{f}$, since q does not hold there and also the formula itself does not hold in the context. In the context $\mathbf{ft}$, the state c3 satisfies q and $\mathbf{E}\top\mathbf{U}p$ (cf. Table 3.3). Therefore, the formula certainly holds in c3 and c1. The call context of call d2 is $\mathbf{ft}$ in both

Table 3.4.: Validity of $\mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}p)$ in $\mathcal{R}_b$

Context	ff	ft	tf	tt
c1	$\mathbf{t}(3)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$
c3	$\mathbf{f}(1)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$
d1	$\mathbf{t}(2)$	$\mathbf{t}(2)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$
d3	$\mathbf{f}(1)$	$\mathbf{f}(1)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$

contexts **ff** and **ft** (cf. Table 3.4 for the first value **f** and Table 3.3 for the latter value **t**). Under this context, the initial state c1 of C is labeled **t**, therefore the formula holds in both contexts in the predecessor location of d2 which is d1. These values can be used to solve c1 for context **ff** to **t**, since the call c2 now has a **t** assumption.

Table 3.4 already shows that three iterations are necessary to label all locations, because the contexts carry transitive dependencies. Thus, in this example, it can be assumed that the number of contexts grows with the formula depth and the number of iterations needed grows with it. This assumption will be confirmed in the next step.

Now consider the third nested formula $\varphi = \mathbf{E}\top\mathbf{U}(p \wedge \mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}p))$. Again the context collection starts at $(C, \mathbf{fff})$. Because the context collection is analogous to the former ones, only the sequence of visited context pairs is shown here:

$$(C, \mathbf{fff}) \longrightarrow (D, \mathbf{mff}) \longrightarrow (C, \mathbf{mft}) \longrightarrow (D, \mathbf{mtt}) \longrightarrow (C, \mathbf{mtt})$$

The contexts can be inferred by looking up the second value from Table 3.4 and the third from Table 3.3. The first value is always **m**, except for the initial context. The last pair $(C, \mathbf{mtt})$ would trigger the collection of $(D, \mathbf{mtt})$ which is already collected. Since every visited context is collected with **t** and **f** as topmost context assumption, this sequence yields a set Ξ consisting of 10 context pairs. The sequence also shows how the formula validity depends on the context: In the initial context $(C, \mathbf{fff})$, no subformula holds and thus also the call of D yields a context $(D, \mathbf{mff})$ in which no subformula holds. Since D contains a p, the lowermost assumption then switches to **t** in the following context $(C, \mathbf{mft})$. Since C contains a q, the second context assumption now also changes to **t** in the context $(D, \mathbf{mtt})$. Table 3.5 depicts the labeling results for the collected contexts.

Table 3.5.: Validity of $\mathbf{E}\top\mathbf{U}(p \wedge \mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}p))$ in $\mathcal{R}_b$

Context	fff	fft	ftt	tff	tft	ttt
c1	$\mathbf{t}(4)$	$\mathbf{t}(2)$	$\mathbf{t}(2)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$
c3	$\mathbf{f}(1)$	$\mathbf{f}(1)$	$\mathbf{f}(1)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$
d1	$\mathbf{t}(3)$	-	$\mathbf{t}(1)$	$\mathbf{t}(1)$	-	$\mathbf{t}(1)$
d3	$\mathbf{f}(1)$	-	$\mathbf{t}(1)$	$\mathbf{t}(1)$	-	$\mathbf{t}(1)$

Fields which have no validity in them are ones which form a context pair that was not collect, like, for example, the pair $(D, \mathbf{tft})$. Again, the three columns on the right are trivially decided to **t**, since the formula itself holds in the context. For the context **ftt**, the locations d1 and d3 can be labeled **t**, because both subformulae, that is p and $\mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}p))$, hold in d3 in this context. The state c3 can be labeled $\mathbf{f}$ in all three contexts on the left, because the formula itself does not hold in the context and q is not reachable from this state. The state d3 can be labeled $\mathbf{f}$ in the context $\mathbf{fff}$, because neither the formula itself holds in the context, nor does the subformula $\mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}p))$ (cf. Context $\mathbf{ff}$ in Table 3.4). All these labels can be assigned in the first iteration. The remaining labels for c1 and d1 depend on the assumption about the calls c2 and d2, respectively, and thus cannot be assigned that early. In the context $\mathbf{ftt}$, the call context of call c2 is $\mathbf{ftt}$ (cf. value of location c3 in Tables 3.5, 3.4, and 3.3 under context $\mathbf{ftt}$, $\mathbf{tt}$, and $\mathbf{t}$, respectively). Since d1 is labeled $\mathbf{t}$ in this context, the assumption about call c2 and thus the label for c1 can be decided to $\mathbf{t}$ in that context. In context $\mathbf{fft}$, the call c2 has also the call context $\mathbf{ftt}$ allowing to label c1 with $\mathbf{t}$ in this context, as well. This value can be used to solve d1 in context $\mathbf{fff}$ in the third iteration. In this context, the call d2 has the call context $\mathbf{fft}$ which leads to a $\mathbf{t}$ assumption and thus a $\mathbf{t}$ label of d1. This label finally decides the call context of c2 under context $\mathbf{fff}$, which is also $\mathbf{fff}$, to $\mathbf{t}$, thus ultimately labeling c1 in the initial configuration with $\mathbf{t}$.

Again, the conclusion about the result for further **EF** formulae is easy: In this case, the number of contexts will grow linearly and so will the number of iterations needed in the labeling algorithm, because the “context chain” necessary to label the initial context gets one step longer with each nested **EF** formula.

The examples show that similar RKSs can lead to quite different results in the algorithm. They also show that the worst case of exponential growth of context pairs with respect to formula depth is not common: In the first example, the number of context pairs did not grow at all, while it showed linear growth in the second example.

3.7. Differences to the Basic Approach

This section elaborates how this algorithm differs from the basic algorithm and why the modifications were performed. When directly comparing this algorithm with the publication of Fehnker et al. [44], there are many notational and nomenclature differences. For example, boxes are named calls in this thesis. These differences are omitted here.

The most obvious modification, which was also described in the first section of this chapter, is that this algorithm does not build a new RKS after checking a formula φ, but saves validity information for φ in the knowledge base κ. The basic algorithm, encodes the validity of φ in the labels of the resulting RKS. In addition, it also encodes the call contexts in the call mapping of the resulting RKS and duplicates sub-structures when they are called in different call contexts. The main reason for dropping this encoding was that the building of an RKS does not work in the incremental refinement presented in the next chapter: Since there is no more checking for one formula, it is impossible to build new RKSs since different formulae must be checked on the same RKS at a time. In addition, the approach using a knowledge base seems more natural than encoding the call context in the call structure of the RKS: A lookup of $\kappa(l, \theta_\varphi)$ can be interpreted easily as: Whenever the sub-structure in which l resides is called with a call stack that satisfies θ_φ, then the validity of φ at location l is $\kappa(l, \theta_\varphi)$. In contrast a label φ in the resulting RKS of the basic algorithm is harder to interpret. For example, consider a sub-structure S which was split into many sub-structures by the basic algorithm. Let one of these sub-structures be S'.

Now, it is not obvious which call stacks S' represents. A call stack σ in the resulting RKS ending at S' can be searched and it can be argued that this call stack represents a member of the equivalence class of call stacks represented by this sub-structure. By inferring the labels at the successor location of the topmost call of that call-stack, the call context can also be inferred. However, this is not as straight forward as in the case of the knowledge base approach.

Another modification is that $\varphi \equiv \mathbf{EX}\psi$ is no longer solved by the *labelByFixpoint* function, or *subcheck*, how this function is called in the basic algorithm. This modification was made, because solving **EX** with fixed point iteration is conceptually disputable: **EX** does not need a fixed point iteration, it can always be solved in the first step. It also does not need any $\mathbf{m}$-labels, because it does not need any information about the validity of φ but only about the validity of ψ, which is fully available. The decision to leave out $\mathbf{EX}\psi$ from the fixed point algorithm makes it more concise by allowing to focus on the important cases. The first simplification is that no formula to solve **EX** in the associated Kripke structure is needed anymore. The second one is that the label $\ulcorner\psi\urcorner$ in the associated Kripke structure gets a unique semantics: It encodes the validity of φ. In contrast, in the basic approach, the label $p^{\mathbf{t}}$ either encodes the validity of φ in case of $\varphi \equiv \mathbf{EG}\psi \mid \mathbf{E}\psi\mathbf{U}\chi$ or of ψ in case of $\varphi \equiv \mathbf{EX}\varphi$, which makes comprehending the algorithm harder.

The summaries in the basic algorithm, which are comparable to the knowledge base in this algorithm, encode assumptions α for calls and guarantees ω for locations. In contrast, the knowledge base only encodes guarantee information for locations. The reason for this is that the assumptions for calls follow directly from the guarantees of locations. Therefore, the algorithm in this thesis always computes the assumptions on-the-fly instead of pre-calculating and storing them in the knowledge base explicitly. While such a pre-calculation is indeed good for performance (and is conducted in the implementation) it is of no conceptual gain and makes reasoning about the algorithm harder. Therefore, it was dropped.

Another change to the basic approach is that the definition of RKSs has been restricted further: In the definition in this thesis, a call may have only one successor location while the definition of Fehnker et al. allows more than one. This change was made purely for simplicity reasons as a single successor location makes the algorithm easier to understand and prove and does not decrease the expressiveness of the model. When implementing the algorithm, it is easy to drop this restriction. This was done in the implementation of XMV which allows more than one successor location.

The final difference is the definition of the RKS $\mathcal{R} = (\mathbb{S}, S^{\text{in}})$ as a set of sub-structures and a designated initial structure instead of the definition $\mathcal{R} = S_1, \ldots, S_n$ as a sequence of sub-structures. As a consequence, there is no specified ordering over the sub-structures anymore, which does not carry any information anyway. The call mapping ν now maps from call directly to sub-structure instead of mapping from call to sub-structure index. The reason for this modification is that the formal definition of the operations *split* and *merge* is very cumbersome when having to maintain indexes. Since these operations where not defined formally in [44], this was no issue for Fehnker et al., but an issue for this thesis.

Chapter 4.
Incremental Refinement

This chapter proposes an incremental refinement of the algorithm from the previous chapter. The aim of this incremental refinement is to leverage the fact that local model checking is usually sufficient, that is, the model checking with respect to the initial configuration. The algorithm from the previous chapter always performs global model checking, labeling all reachable configurations of the RKS. The incremental refinement carefully tries to find the smallest set of context pairs which are necessary to label the initial configuration correctly and performs the model checking only with these pairs instead of all reachable pairs. The set computed by the algorithm is not completely minimal but rather an overestimation which is believed to be still sufficiently small to yield a noticeable performance increase. The chapter starts with an overview over the incremental refinement, followed by a brief comparison to the algorithm from the previous chapter. Then, the incremental algorithm is described in detail. Finally, its correctness is proven.

4.1. Overview

The algorithm shown in the previous chapter performs labeling with respect to a formula φ in two steps: First, the reachable context pairs are collected and next, these context pairs are labeled thoroughly. For large RKSs which consist of thousands of sub-structures, the set of all reachable contexts may be quite large and hence the labeling of all these contexts might take long. Since the primary target of model checking is to label the initial configuration of the RKS correctly, it is not necessary that all locations in all reachable contexts are actually labeled. Instead, it suffices to label only as many locations so that the label for the initial location can be inferred. This is what the incremental refinement achieves. For example, formulae of the form $\neg\psi$ can always be solved locally in the initial configuration. In contrast, the formula $\mathbf{EG}\psi$ is not always solved that easily.

While the basic approach always collects all reachable contexts to label them, the incremental refinement collects only contexts which are necessary to label the initial configuration. By exploring the call context graph and stopping the exploration whenever the validity of φ can be decided locally for the explored context, a set of contexts is explored which cannot be decided locally. Once such set is found, it is checked with the *labelByFixpoint* algorithm presented in the previous chapter, which is only slightly adapted.

The basic algorithm is strictly separated into the steps "collect contexts" and "label these contexts". No such separation is possible in the incremental refinement, because it must be tried to label the currently explored context in order to decide whether this context can be labeled locally or more contexts have to be explored.

In addition, the algorithm no longer works strictly on one formula φ at a time. Instead, it is possible that while checking φ, the algorithm must "go back" to labeling a subformula ψ. For example, consider the formula $\varphi \equiv \mathbf{EG}p$. The proposition p can always be solved locally without looking at other contexts than the initial one. So a recursive call to the labeling algorithm would only label the initial context with respect to p. It is, however, possible that the formula $\mathbf{EG}p$ cannot be solved in the initial context only. In this case, further contexts have to be explored and labeled. These contexts do not have a labeling for p yet, so, the algorithm must label p for these contexts before it can label $\mathbf{EG}p$ for them.

The incremental labeling of a call context pair (S, θ_φ) with respect to a formula φ is conducted in three steps. Each of them must only be performed if the one before is not able to yield a result:

1. It is tried to label the locations of S in context θ_φ locally, using the already computed values in the knowledge base κ. If a result is obtained (i.e., the location l^{in} is either labeled $\mathbf{t}$ or $\mathbf{f}$), then the checking for this context pair is finished without checking further ones. This is called *local checking*. If the initial location is still labeled $\mathbf{m}$ (i.e., not yet labeled) after the local check, this can have two reasons: First, labels for subformulae of φ might still be missing. Second, some call assumptions might still be $\mathbf{m}$ and the labeling of the initial location might depend on these assumptions. The algorithm checks for these two possible reasons in the remaining steps.

2. It is checked whether the knowledge base is fully labeled in the current context with respect to the subformulae of φ. If it is not, then the algorithm first labels all locations in the current context with respect to the subformulae of φ. For example, in the case of $\varphi \equiv \mathbf{EG}\psi$, the algorithm tries to label all locations in the current context with respect to the formula ψ. This is called *vertical checking*, since the algorithm "descends vertically down" the structure of the formula. After the vertical checking has been performed, another local check attempt can be made, since the additional labels with respect to subformulae of φ might suffice to label the initial location with respect to φ.

3. If the initial location is not labeled yet after the vertical checking has been performed thoroughly, then the result must depend on the assumptions made about calls. If this is the case, the algorithm must pick a call $c \in C(S)$ from the calls of the sub-structure of the current context and try to label the initial location of the sub-structure $S_{\leftarrow c}$ called by that call under the call context *cc* of that call. This step is called *horizontal checking* and is similar to the graph exploration of the basic algorithm.

Of course, the horizontal checking can run into a loop of sub-structures which call each other and cannot be decided locally. In such situation, the set of sub-structures which take part in such a loop must be decided with the usual *labelByFixpoint* algorithm. To gather the sub-structures which have to be checked together because they take part in such a call-loop, the set of visited but not yet decided context pairs is kept while performing the horizontal checking. This is equal to the set Ξ which gathered the reachable context pairs in the basic algorithm. The checking starts with the initial sub-structure S^{in} in the initial context *icc*.

Note that labeling must be performed in two different ways: In the steps 1 and 3, it is sufficient to label the initial location. This is called *partly* checking of a context. In step 2, all locations must be labeled. This is called *fully* checking of a context. Also note that the formulae $\top$, $p \in AP$, $\neg\psi$, and $\psi \vee \chi$ can always be decided locally without considering any further contexts except for the initial one and the formula $\mathbf{EX}\psi$ can be decided with looking at most "one step further". For these formulae, the incremental refinement always yields a huge performance benefit. For the operators **EG** and **EU**, the benefit of the incremental refinement depends on whether local paths satisfying the formulae exist.

4.2. Differences to the Non-Incremenal Algorithm

The biggest difference between the algorithms, which is also the advantage of the incremental refinement and the reason why it is proposed, is that the incremental version checks only as many context pairs as necessary while the basic approach always checks all reachable context pairs. This property can lead to large performance increases, especially for formulae which can be decided locally. For example, consider the formula $\Phi = \mathbf{EG}p$ and a huge RKS of thousands of sub-structures. If the model checking is used for static program analysis, where each function is mapped to a sub-structure, this is a quite realistic scenario for larger programs. If the initial sub-structure S^{in} already contains a local path which is labeled p in each location, then the formula Φ can be decided locally in S^{in} without checking any other sub-structure. While the incremental algorithm would do that, the algorithm presented in the previous section would check all reachable context pairs.

A conceptual difference between the algorithms is that the incremental algorithm is often forced to check different formulae for different sub-structures at a time, while the basic approach always has one formula which is currently being checked and only checks further formulae once this formula has been checked thoroughly. This property of the incremental approach originates from its "lazy" checking of sub-structures: Consider a formula φ which can be decided in the initial sub-structure S^{in}. Then κ is labeled correctly with respect to φ for all locations in S^{in}, but not labeled yet for other sub-structures. Afterwards, if a formula which uses φ as subformula, like $\mathbf{EG}\varphi$, requires the checking in additional sub-structures, then φ has to be checked for these sub-structures. Thus, model checking has to be performed with respect to φ even after the algorithm is finished with the initial checking of φ.

4.3. Algorithmic Details

A main feature of the incremental refinement is to stop whenever a sub-structure is sufficiently labeled. This notion is defined formally as follows:

Definition 4.3.1 (Fully Labeled Context). *Let κ be a knowledge base, S a sub-structure, φ a CTL formula, and θ_φ a call context under which S is called with respect to φ. κ is said to be* fully labeled *with respect to (S, θ_φ) if all locations are labeled either* $\mathbf{t}$ *or* $\mathbf{f}$ *for this context pair:*

$$\forall l \in L(S) \,.\, \kappa(l, \theta_\varphi) \in \{\mathbf{t}, \mathbf{f}\}$$

Furthermore, κ is said to be partly labeled *with respect to (S, θ_φ), if the initial location is labeled:*

$$\kappa(l^{\text{in}}(S), \theta_\varphi) \in \{\mathbf{t}, \mathbf{f}\}$$

κ is said to be not sufficiently labeled *if it is not partly labeled.*

The function isLabeled : $\mathbb{K} \times \mathbb{S} \times \Theta \times \{\mathsf{fully}, \mathsf{partly}\} \rightarrow \mathbb{B}$ *returns whether a knowledge base is labeled fully or partly:*

$$\begin{aligned} \textit{isLabeled}(\kappa, S, \theta, \mathsf{partly}) = \mathbf{t} &\Leftrightarrow \text{“}\kappa \textit{ partly labeled with respect to } (S, \theta)\text{”} \\ \textit{isLabeled}(\kappa, S, \theta, \mathsf{fully}) = \mathbf{t} &\Leftrightarrow \text{“}\kappa \textit{ fully labeled with respect to } (S, \theta)\text{”} \end{aligned}$$

As mentioned in the overview, no specific formula φ is labeled at a time. Therefore, there is no more need for a function *labelOneFormula*. Instead only a *label* function is used, which is shown in Algorithm 4.1.

Algorithm 4.1 *label* : $\mathbb{K} \times \mathbb{S} \times \Theta \times \{\mathsf{fully}, \mathsf{partly}\} \rightarrow \mathbb{K}$

```
fun label(κ, S, θ_φ, f) =
 1: if isLabeled(κ, S, θ_φ, f) then
 2:     return κ
 3: end if
 4: Ξ ← collectContexts(κ, S, f)
 5: Ξ ← {(S′, θ′) ∈ Ξ | ¬isLabeled(κ, S′, θ′, partly)}
 6: if f = fully ∧ ¬isLabeled(κ, S, θ_φ, f) then
 7:     Ξ ← Ξ ∪ (S, θ_φ)
 8: end if
 9: if Ξ ≠ ∅ then    //Local checking not sufficient?
10:     κ ← labelByFixpoint(κ, Ξ, φ)
11: end if
12: return κ
```

The label function has a different signature than in the basic algorithm. It takes a knowledge base κ, a sub-structure S and a call context θ_φ. The result is an updated knowledge base which is labeled with respect to (S, θ_φ). The final parameter f determines if the labeling should stop once the the initial location is is labeled (partly) or whether all locations in that sub-structure must under context θ must be labeled (fully).

The first check done by the algorithm is to check whether the input knowledge base is already sufficiently labeled. If it is, the algorithm returns the unchanged knowledge base. The rest of the function is comparable to *labelOneFormula* of the basic algorithm: First the context to be labeled are collected (line 4), then they are labeled (line 10). Of course, there are also differences. As mentioned, most of the labeling is already done in *collectContexts*. Especially local checking is already performed in this step. This always yields a result for non-temporal formulae and for **EX**. Local checking can only fail in case of **EG** and **EU**. After the contexts are collected, already partly labeled contexts are removed from the set Ξ (line 5). Now, Ξ only contains contexts which could not be labeled partly. This is exactly the set of contexts which must be labeled using the usual fixed point algorithm. If Ξ is not empty (i.e., there were insufficiently labeled contexts), then the fixed point algorithm is executed on this set (line 10). The *labelByFixpoint* function is the one from the basic algorithm, as depicted by Algorithm 3.4 on page 48, with minor differences described in the next section. Finally, the updated knowledge base is returned. If the check is to be conducted fully but the context pair to be checked is not labeled fully after the context collection, yet, then the pair is explicitly added to Ξ again (lines 6 to 8). This is necessary, because line 5 would remove the pair from Ξ, if it is partly labeled. This would leave the pair only checked partly, while fully checking is required. Other pairs in Ξ can always safely be removed if they are partly labeled, because they were collected by horizontal exploration which is always only performed partly.

The function *collectContext*, as presented in Algorithm 4.2, differs from the equally named function from the basic algorithm, because it also performs the local checking and stops the context-graph-exploration whenever a context pair can be decided locally.

Algorithm 4.2 *collectContexts* : $\mathbb{K} \times \mathbb{S} \times \Theta \times \{\mathsf{fully}, \mathsf{partly}\} \to \wp(\mathbb{S} \times \Theta)$

```
fun $collectContexts(\kappa, S, \theta, f) = \lambda(\emptyset, \kappa, S, \theta, f)$
fun $\lambda(\Xi, \kappa, S, \theta \equiv \langle\eta\rangle\tau, f) =$
 1: $\Xi \leftarrow \Xi \cup ($if $\tau \equiv \top$ then $\{(S, \langle\mathbf{t}\rangle\tau)\}$ else $\{(S, \langle\mathbf{t}\rangle\tau), (S, \langle\mathbf{f}\rangle\tau)\})$
 2: $\kappa \leftarrow labelLocallyAndVertically(\kappa, S, \theta, f)$
 3: for all $c \in C(S)$ do
 4:     break when $isLabeled(\kappa, S, \theta, f)$
 5:     $\theta_c \leftarrow cc(\kappa, c, \theta)$
 6:     $S_c \leftarrow \nu_S(c)$
 7:     if $(S_c, \theta_c) \notin \Xi \wedge \neg isLabeled(\kappa, S_c, \theta_c, \mathsf{partly})$ then
 8:         $\lambda(\Xi, \kappa, S_c, \theta_c, \mathsf{partly})$   //Horizonal labeling
 9:         $\kappa \leftarrow labelLocally(\kappa, S, \theta_\varphi)$   //Try locally again
10:     end if
11: end for
12: return $\Xi$
```

The first difference is the signature which now contains an additional parameter f stating whether the checking should be conducted fully or partly. The next change is that the currently visited context pair is labeled locally and vertically (line 2). This might already yield a knowledge base which is labeled sufficiently. To stop as soon as this is the case, the first statement in the loop (line 4) leaves the loop as soon as the currently visited context pair is sufficiently labeled. If this is the case in the first iteration, no call is actually processed and the function returns immediately. The next difference is that the exploration of calls is not done for calls which are already partly labeled (line 7). It is sufficient to label these calls only partly, because the only important value is the one of the initial state which is used as assumption about the call. If a not sufficiently labeled call is found, it is explored. A partly labeled context pair has a correctly labeled initial location and is thus sufficient. After the horizontal checking, the algorithm tries to label the current context locally again (line 9). This is important, because the horizontal checking might have labeled the initial location of the explored context or even of another context which was transitively checked. If this labels the context sufficiently, the loop will be left in the next iteration (line 4). This ensures that no unnecessary further exploration is done. The rest of the function is identical with the basic algorithm.

As the name states, the function *labelLocallyAndVertically*, as depicted in Algorithm 4.3, conducts the local and vertical checking of a context pair.

Algorithm 4.3 *labelLocallyAndVertically* : $\mathbb{K} \times \mathbb{S} \times \Theta \times \{\mathsf{fully}, \mathsf{partly}\} \to \mathbb{K}$

```
fun $labelLocallyAndVertically(\kappa, S, \theta_\varphi \equiv \langle\eta\rangle\tau, f) =$
 1: if $\neg isLabeled(\kappa, S, \theta_\varphi, f)$ then
 2:   $\kappa \leftarrow labelLocally(\kappa, S, \theta_\varphi)$   //Label locally
 3:   if $\neg isLabeled(\kappa, S, \theta_\varphi, f)$ then
 4:     $\kappa' \leftarrow \kappa$
 5:     if $(\tau \equiv \neg\theta_\psi \mid \mathbf{EX}\theta_\psi \mid \mathbf{EG}\theta_\psi \mid \theta_\psi \vee \theta_\chi \mid \mathbf{E}\theta_\psi\mathbf{U}\theta_\chi) \wedge \neg isLabeled(\kappa, S, \theta_\psi, \mathsf{fully})$ then
 6:       $\kappa \leftarrow label(\kappa, S, \theta_\psi, \mathsf{fully})$   //Vertical labeling w.r.t. $\psi$
 7:     end if
 8:     if $(\tau \equiv \theta_\psi \vee \theta_\chi \mid \mathbf{E}\theta_\psi\mathbf{U}\theta_\chi) \wedge \neg isLabeled(\kappa, S, \theta_\chi, \mathsf{fully})$ then
 9:       $\kappa \leftarrow label(\kappa, S, \theta_\chi, \mathsf{fully})$   //Vertical labeling w.r.t. $\chi$
10:     end if
11:     if $\kappa' \neq \kappa$ then
12:       $\kappa \leftarrow labelLocally(\kappa, S, \theta_\varphi)$   //Try locally again
13:     end if
14:   end if
15: end if
16: return $\kappa$
```

If the context is already sufficiently labeled, the function returns immediately (line 1). Otherwise, the function first tries to label the context locally (line 2). After this try, it is checked if the local checking already achieved a sufficient labeling (line 3). If this is the case, the function returns immediately. If the local labeling did not suffice, the next step is to do the vertical labeling, which means the labeling with respect to subformulae. As mentioned, vertical labeling always has to be performed fully to ensure that all locations are labeled with respect to the subformulae of φ. Of course, the labeling is only done for a subformula if the knowledge base is not yet fully labeled with respect to that formula. The conditional vertical labeling is performed in the lines 5 to 10. Finally, if any vertical labeling was conducted, the algorithm tries to label the context locally again (line 12). This must be done, because the vertical labeling of subformulae of φ could yield new results which could be sufficient to label the context sufficiently with respect to φ.

Example: Consider the RKS $\mathcal{R} = (\{A, B, C, D, \ldots\}, A)$ over propositions $AP = \{p, q\}$. The sub-structures A, B and C of this RKS are depicted in Figure 4.1. The sub-structure D and possible further sub-structures called by D are omitted. The RKS is to be model checked with respect to the formula $\varphi = \mathbf{E}(\mathbf{EG}p)\mathbf{U}q$. Figure 4.2 depicts the order in which the incremental checking process is conducted. In this figure, a rectangle depicts a formula to be checked for a sub-structure. An arrow depicts the invocation of another check. Vertical arrows represent a vertical labeling invocation while horizontal arrows represent a horizontal labeling invocation. A rectangle highlighted in yellow depicts a partly labeling while white rectangles are labeled fully. Note that the checking is usually performed on context pairs. For simplicity reasons, the example context was left out in the figure. The example is constructed so that the context never matters in it, so only the sub-structure of

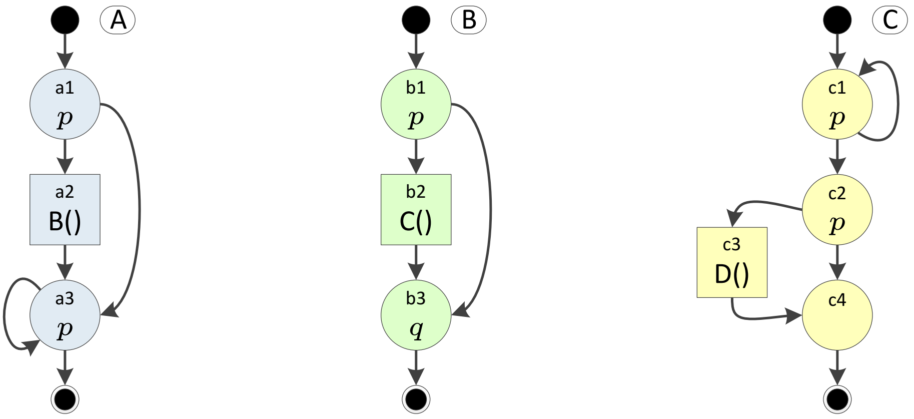


Figure 4.1.: Sub-structures of an example RKS

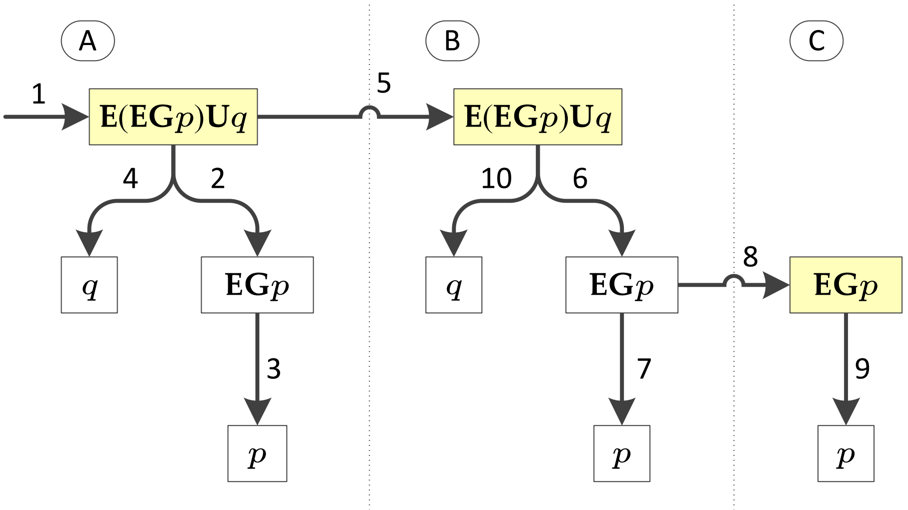

A
B
C
1
5
$\mathbf{E}(\mathbf{EG}p)\mathbf{U}q$
$\mathbf{E}(\mathbf{EG}p)\mathbf{U}q$
4
2
10
6
q
$\mathbf{EG}p$
q
$\mathbf{EG}p$
8
$\mathbf{EG}p$
3
7
9
p
p
p

Figure 4.2.: Order of the checking process of the RKS from Figure 4.1 with respect to formula $\mathbf{E}(\mathbf{EG}p)\mathbf{U}q$

a context pair is important.

The checking starts by checking the initial sub-structure A against the full formula $\varphi = \mathbf{E}(\mathbf{EG}p)\mathbf{U}q$ (Arrow 1). The local check fails due to missing subformula labels. Therefore, a check of A against $\mathbf{EG}p$ is performed (2). This check also fails due to missing labels for p, so A is labeled with p first (3). With the labels for p, the sub-structure can be fully labeled with respect to $\mathbf{EG}p$, because the self-loop in a3 ensures that $\mathbf{EG}p$ always holds in all locations of A, regardless of the assumption about call a2. Thus, no other sub-structure has to be regarded for this formula and the local checking succeeds. Next, the second subformula q is checked on sub-structure A (4). Now, A is fully labeled with respect to all subformulae of φ. However, the formula itself still cannot be decided, because no q label is in A and thus the validity depends on the assumption about call a2. Therefore, a horizontal check of B with respect to φ is invoked (5). Since B has no labels for any formula yet, the same vertical checking is performed as for A (6,7,9). In contrast to A, B, or more precisely the location b1, cannot be decided locally with respect to $\mathbf{EG}p$. Instead, the validity depends on the call b2. Thus, a vertical checking of C with respect to $\mathbf{EG}p$ has to be conducted (8). After labeling p for this sub-structure (9), $\mathbf{EG}p$ can be decided locally for it. Here, it is especially important that the horizontal check is conducted partly: c1 can be decided locally due to its self-loop, but c2 cannot — it depends on assumptions about c3. Thus, a full labeling would require a horizontal check of D. Because the labeling of c1 suffices for the partly check, no further horizontal checking is necessary. After the partly labeling of C is accomplished, B can be labeled with respect to $\mathbf{EG}p$ and also with respect to φ: b1 and b3 both satisfy φ due to the path $\mathtt{b1}, \mathtt{b3}$ satisfying $(\mathbf{EG}p)\mathbf{U}q$. Thus, the assumption about B becomes true and a1 can also be labeled true, due to the path $\mathtt{a1}, \mathtt{b1}, \mathtt{b3}$. Now the checking is finished.

The example shows that the checking stops at C without checking D. Now consider that the RKS has thousands of further sub-structures called transitively from D. In this case, the incremental refinement would have yielded a drastic speed-up.

The local labeling of κ with respect to a context and a formula is performed by the function *labelLocally* as depicted by Algorithm 4.4. The exact meaning of local labeling is to infer labels of a sub-structure call context pair (S, θ_φ) without labeling any other sub-structures or call contexts. The function is basically equal to *computeLabels* of the basic algorithm, as depicted by Algorithm 3.3 on page 46. Note that the function β is the same as for *computeLabels* (cf. Equation 3.6 on page 46). The most important difference is that *labelLocally* does not loop over all contexts in a set Ξ and labels all of them. Instead, it labels exactly one pair (S, θ_φ), because this is the definition of local labeling. The only further difference is that the function calls *deduceLabels* instead of *labelByFixpoint* for the cases **EG** and **EU**. The reason for this is again the local labeling: While *labelByFixpoint* needs a whole set of contexts, *deduceLabels* only needs one context pair to be labeled. The *deduceLabels* function is the one of the basic algorithm (cf. 3.5 on page 50) with the difference that it must be able to label a formula φ in the presence of $\mathbf{m}$ values for subformulae of φ, because it might be the case that these subformulae are not labeled vertically yet. This is described in the next section.

Algorithm 4.4 *labelLocally* : $\mathbb{K} \times \mathbb{S} \times \Theta \to \mathbb{K}$

```
fun labelLocally($\kappa, S, \theta_\varphi \equiv \langle\eta\rangle\tau$) =
    if $\tau \equiv \top$ then
        for all $l \in L(S)$ do $\kappa(l, \theta_\varphi) \leftarrow \mathbf{t}$
    else if $\tau \equiv p \in AP$ then
        for all $l \in L(S)$ do $\kappa(l, \theta_\varphi) \leftarrow (p \in \mu_S(l))_3$
    else if $\tau \equiv \neg\theta_\psi$ then
        for all $l \in L(S)$ do $\kappa(l, \theta_\varphi) \leftarrow \neg\kappa(l, \theta_\psi)$
    else if $\tau \equiv \theta_\psi \vee \theta_\chi$ then
        for all $l \in L(S)$ do $\kappa(l, \theta_\varphi) \leftarrow \kappa(l, \theta_\psi) \vee \kappa(l, \theta_\chi)$
    else if $\tau \equiv \mathbf{EX}\theta_\psi \equiv \mathbf{EX}\langle\eta_\psi\rangle\tau_\psi$ then
        for all $l \in L(S) \setminus \{l^{\mathrm{out}}(S)\}$ do
            $\kappa(l, \theta_\varphi) \leftarrow \beta(\kappa, c, \theta_\psi)$
        end for
        $\kappa(l^{\mathrm{out}}(S), \theta_\varphi) \leftarrow \eta_\psi$
    else if $\tau \equiv \mathbf{EG}\theta_\psi \mid \mathbf{E}\theta_\psi\mathbf{U}\theta_\chi$ then
        $\kappa \leftarrow$ deduceLabels($\kappa, S, \theta_\varphi$)
    end if
    return $\kappa$
```

4.4. EG and EU Labeling in Presence of Subformula m Values

Due to the incremental refinement, it may be the case that a context pair is to be labeled with respect to a formula φ while not all locations in that context are labeled with respect to the subformulae of φ. Thus, the definition of the associated Kripke structure $\mathcal{K}$ has to be adapted to incorporate $\mathbf{m}$ and $\mathbf{t}$ versions of subformula labels. For example, while the basic algorithm only used $\ulcorner\psi\urcorner$ to denote the validity of the subformula ψ in a location, the incremental refinement must use $\ulcorner\psi\urcorner$ or $\ulcorner\psi?\urcorner$ to denote that ψ holds or it is unknown whether ψ holds, respectively.

Definition 4.4.1 (Incremental Associated Kripke Structure)**.** *Let $S = (L, l^{\mathrm{in}}, l^{\mathrm{out}}, C, \delta, \mu, \nu)$ be a sub-structure, κ be a knowledge base and $\theta \equiv \langle\eta\rangle\mathbf{EG}\theta_\psi \mid \langle\eta\rangle\mathbf{E}\theta_\psi\mathbf{U}\theta_\chi$ be a call context. Let $AP = \{\ulcorner\psi\urcorner, \ulcorner\psi?\urcorner, \ulcorner\chi\urcorner, \ulcorner\chi?\urcorner, \ulcorner\varphi\urcorner, \ulcorner\varphi?\urcorner\}$ be a set of atomic propositions.*

Let $\alpha : \mathbb{T} \times AP \times AP \to \wp(AP)$ be a function which is defined as follows:

$$\alpha(t, p_\mathbf{t}, p_\mathbf{m}) = \begin{cases} \{p_\mathbf{t}\} & \text{if } t = \mathbf{t} \\ \{p_\mathbf{m}\} & \text{if } t = \mathbf{m} \\ \emptyset & \text{otherwise} \end{cases}$$

The incremental associated Kripke structure *$\mathcal{K}_{\mathrm{inc}}(\kappa, S, \theta) = (\mathcal{S}, I, \delta, \mu)$ is a Kripke structure over the set of atomic propositions AP with*

- *the set of states $\mathcal{S} = L \cup C \cup \{\omega\}$*
- *the set of initial states $I = \{l^{\mathrm{in}}\}$*
- *the transition relation $\delta = \delta \cup \{(l^{\mathrm{out}}, \omega), (\omega, \omega)\}$*

- *the labeling function $\mu : \mathcal{S} \to \wp(AP)$ which is defined as follows:*

$$
\begin{aligned}
\mu(l) &= \begin{cases} \alpha(\kappa(l,\theta_\psi), \ulcorner\psi\urcorner, \ulcorner\psi?\urcorner) & \textit{if } \theta \equiv \langle\eta\rangle \mathbf{EG}\theta_\psi \\ \quad \alpha(\kappa(l,\theta_\psi), \ulcorner\psi\urcorner, \ulcorner\psi?\urcorner) & \multirow{2}{*}{} \textit{if } \theta \equiv \langle\eta\rangle \mathbf{E}\theta_\psi \mathbf{U}\theta_\chi \\ \cup\, \alpha(\kappa(l,\theta_\chi), \ulcorner\chi\urcorner, \ulcorner\chi?\urcorner) & \end{cases} && \textit{for all } l \in L \\
\mu(c) &= \alpha\left(\kappa(l^{\text{in}}(S_{\leftarrow c}), cc(\kappa, c, \theta)), \ulcorner\varphi\urcorner, \ulcorner\varphi?\urcorner\right) && \textit{for all } c \in C \\
\mu(\omega) &= \alpha(\eta, \ulcorner\varphi\urcorner, \ulcorner\varphi?\urcorner)
\end{aligned}
$$

The definition of the incremental associated Kripke structure $\mathcal{K}_{\text{inc}}$ resembles the one of the basic algorithm (cf. Definition 3.5.1 on page 48), but provides $\mathbf{t}$ and $\mathbf{m}$ labels for the subformulae ψ and χ. The formulae which are checked on this Kripke structure must also be adapted to incorporate the new labels for subformulae.

Algorithm 4.5 *deduceLabels* : $\mathbb{K} \times \mathbb{S} \times \Theta \to \mathbb{K}$

```
fun deduceLabels($\kappa, S, \theta_\varphi \equiv \langle\eta\rangle\tau$) =
$\varphi_{\mathbf{t}} \leftarrow$ ( if $\tau \equiv \mathbf{EG}\theta_\psi$ then $\mathbf{EG}\ulcorner\psi\urcorner$ else $\mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\chi\urcorner$)
$\varphi_{\mathbf{t}\vee\mathbf{m}} \leftarrow$ ( if $\tau \equiv \mathbf{EG}\theta_\psi$ then $\mathbf{EG}(\ulcorner\psi\urcorner \vee \ulcorner\psi?\urcorner)$ else $\mathbf{E}(\ulcorner\psi\urcorner \vee \ulcorner\psi?\urcorner)\mathbf{U}(\ulcorner\chi\urcorner \vee \ulcorner\chi?\urcorner)$)
for all $l \in L(S)$ do
  if $\kappa(l, \theta_\varphi) = \mathbf{m}$ then
    if $\mathcal{K}_{\text{inc}}(\kappa, S, \theta_\varphi), l \models \varphi_{\mathbf{t}} \vee \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}\ulcorner\varphi\urcorner$ then
      $\kappa(l, \theta_\varphi) \leftarrow \mathbf{t}$
    else if $\mathcal{K}_{\text{inc}}(\kappa, S, \theta_\varphi), l \models \neg(\varphi_{\mathbf{t}\vee\mathbf{m}} \vee \mathbf{E}(\ulcorner\psi\urcorner \vee \ulcorner\psi?\urcorner)\mathbf{U}(\ulcorner\varphi\urcorner \vee \ulcorner\varphi?\urcorner))$ then
      $\kappa(l, \theta_\varphi) \leftarrow \mathbf{f}$
    end if
  end if
end for
return $\kappa$
```

The modified *deduceLabels* function is shown in Algorithm 4.5. The formula for assigning a $\mathbf{t}$ label has stayed the same. The formula for assigning an $\mathbf{f}$ label now contains the $\mathbf{t}$ and $\mathbf{m}$ versions of the sub-formula propositions. This is obviously the correct replacement: An $\mathbf{f}$ label can be assigned if no path exists where the formulae either surely holds ($\mathbf{t}$) or it is unknown whether the formula holds ($\mathbf{m}$). It is not proven that this labeling is correct here. A proof would be similar to the one for the correctness of the basic *deduceLabels* function, as stated by Lemma 3.5.2.

4.5. Correctness of the Incremental Refinement

Many parts of the incremental refinement still use the same techniques as shown in the previous chapter. Therefore, the proof for these techniques can be used for the incremental refinement, too. The only things which have to be proven are the aspects where the incremental refinement modifies the basic algorithm. These are the following aspects:

1. The algorithm already performs local checking during the context collection. It has to be shown that this labeling does not introduce incorrect labels.

2. After the termination of the context collection and after partly labeled contexts are removed from Ξ, it is checked whether Ξ is empty. If not, then the contexts in the set are labeled with the *labelByFixpoint* function. This function is only defined for **EG** and **EU**. Therefore, it must be proven that all other operators will certainly not yield any unlabeled contexts in Ξ and thus *labelByFixpoint* will never be executed for these operators.

3. The *labelByFixpoint* function only works correctly if all labels for subformulae of φ are known. Otherwise, there could be a path with locations labeled $\mathbf{m}$ which is only labeled $\mathbf{m}$ due to unknown subformulae. It would not be correct to label such path with *decideRemainingMaybes* as done by *labelByFixpoint*.

4. The labeling of **EG** and **EU** is performed by *labelByFixpoint* as in the basic algorithm. However, the set of contexts Ξ considered by the algorithm is not all reachable contexts. It has to be shown that this subset suffices to yield correct results.

The first point and the second point are rather straight-forward but the third point requires some effort. The three points are to be proven one by one. Note that the proofs for the incremental refinement are not conducted as precise as the proofs for the basic algorithm. Nevertheless, they should be formally enough to be verifyable.

Lemma 4.5.1 (Correctness of the Local Labeling)**.** *The local checking performed during the execution of collectContexts does not introduce incorrect labels.*

Proof. The local checking is performed by *labelLocally*, as depicted in Algorithm 4.4. For all operators except **EG** and **EU**, this function is equal to the *computeLabels* function of the basic algorithm. Therefore, the same reasoning can be used to prove the correctness. The only difference is that labels for subformulae might still be missing (i.e., $\mathbf{m}$) in this step. However, it is trivial to see that all operators used will yield correct results even in the presence of $\mathbf{m}$ values.

The only difference to *computeLabels* is that *deduceLabels* is used for **EG** and **EU** formulae. As already said in the section where this function was introduced for the incremental refinement, its correctness is not proven explicitly because this proof is quite similar to the proof for the correctness of the basic *deduceLabels* function, as stated by Lemma 3.5.2. The correctness can be deduced trivially using the reasoning in that proof. □

The next point to be proven is that all operators except **EG** and **EU** are already labeled during the context collection. Therefore, the set Ξ is empty after removing partly labeled contexts and *labelByFixpoint* is not executed for these operators.

Lemma 4.5.2 (*collectContexts* always decides trivial operators)**.** *Let* $\varphi \equiv \top \mid p \in AP \mid \neg\psi \mid \psi \vee \chi \mid \mathbf{EX}\psi$ *be a CTL formula. The function labelByFixpoint will never be called for* φ *when performing the incremental labeling algorithm on this formula.*

Proof. The function *labelByFixpoint* will not be executed by the *label* (Algorithm 4.1), if the set Ξ is empty. This is the case, if all context pairs that were added by *collectContexts* are partly labeled, because then they will be removed from the set at line 5 of Algorithm 4.1.

Formulae of the form $\varphi \equiv \top \mid p \in AP$ are trivially labeled locally without considering further contexts than the initial one.

The non-temporal formulae $\varphi \equiv \neg\psi \mid \psi \vee \chi$ are also labeled without considering other contexts than the initial one. In this case, the call to *labelLocallyAndVertically* will label the initial context. Therefore, no further context will be added. First, the function *labelLocallyAndVertically* performs a local labeling step. This step might fail due to missing subformula labels. If this is the case, then the context is labeled *fully* with respect to the subformulae of φ and then the local labeling is repeated. This time, the local labeling will certainly label all locations, because the context is labeled fully with respect to all subformulae now.

In case of $\varphi \equiv \mathbf{EX}\psi$, additional contexts might be visited. However, all visited context will be labeled at least partly, because **EX** can always be labeled as long as the validity of ψ is known for called contexts. This is surely the case for all visited contexts, because a full labeling of subformulae is performed during the execution of *labelLocallyAndVertically*. □

For the remaining cases $\varphi \equiv \mathbf{EG}\psi$ and $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$, it must be shown that all contexts in Ξ are fully labeled with respect to ψ and χ, because otherwise, the *labelByFixpoint* algorithm might fail.

Lemma 4.5.3 (Contexts in Ξ are Fully Labeled With Respect to Subformulae)**.** *Let $\varphi \equiv \mathbf{EG}\psi \mid \mathbf{E}\psi\mathbf{U}\chi$ be a CTL formula and κ a knowledge base, as given to the labelByFixpoint(κ, Ξ, φ) operation in Algorithm 4.1. At this point of the algorithm, the knowledge base is always fully decided for all context pairs in Ξ with respect to the formulae ψ and χ (the latter only in case of $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$).*

Proof. By contradiction: consider a pair (S, θ_φ) in Ξ which is not labeled fully with respect to ψ and χ. This pair was inserted and thus visited by *collectContexts*. This function calls *labelLocallyAndVertically* for the visited pair, which first tries a local labeling. If this labeling suffices to label the context pair at least partly, then the method returns. In this case, the pair cannot be in the set Ξ, because it would have been removed after the context collection, since it is partly labeled. Thus, the local labeling could not suffice to label (S, θ_φ) partly. In this case, the *labelLocallyAndVertically* performs a full vertical labeling with respect to all subformulae of φ. Therefore, (S, θ_φ) is fully labeled with respect to all subformulae which contradicts the assumption that it is not. □

The last thing to be proven is that the set Ξ, which is not the set of all reachable context pairs in the incremental refinement, suffices so that the *labelByFixpoint* yields a correct result. This property, which is called check-set completeness, is proposed by Proposition 4.5.7. To prove this proposition, some additional lemmas are used:

Lemma 4.5.4 (Check-Set Isolation)**.** *Let Ξ be the result of the collectContext function, from which the partly decided context pairs have already been removed. For all context pairs (S, θ) and all calls $c \in C(S)$, the context pair $(S_{\leftarrow c}, \theta(c, \theta))$ which depicts the target context pair of c is either in Ξ or is partly labeled:*

$$\forall (S, \theta) \in \Xi, c \in C(S)\,.\, \mathit{isLabeled}(\kappa, S_{\leftarrow c}, \theta(c, \theta), \mathsf{partly}) \vee (S_{\leftarrow c}, \theta(c, \theta)) \in \Xi$$

Proof. Consider an arbitrary pair $(S, \theta) \in \Xi$. Since the pair is in Ξ, it was visited by Algorithm 4.2. In this step, the algorithm has visited (and thus added to Ξ) either all target pairs $(S_{\leftarrow c}, \theta(c, \theta))$ of all calls $c \in C(S)$ or has terminates early if (S, θ) had become labeled fully or partly. If the latter would be the case, then (S, θ) would be removed from Ξ. Since it is assumed that (S, θ) is in Ξ, all target pairs $(S_{\leftarrow c}, \theta(c, \theta))$ must have been visited and are thus in Ξ or have been removed from it if they are decided. □

Lemma 4.5.5 (Partly decided context pairs can be left out). *Let κ^{in} be a knowledge base, φ a CTL formula, and Ξ a set of context pairs obtained from the context collection and removal of partly decided contexts in Algorithm 4.1. Let $(S', \theta'_\varphi) \notin \Xi$ be a context pair for which κ^{in} is partly labeled with respect to φ. Let*

$$\kappa \leftarrow \mathit{labelByFixpoint}(\kappa^{in}, \Xi, \varphi)$$

and

$$\kappa' \leftarrow \mathit{labelByFixpoint}(\kappa^{in}, \Xi \cup \{(S, \theta_\varphi)\}, \varphi)$$

The addition of (S', θ'_φ) does not alter the result of labelByFixpoint with respect to all context pairs in Ξ:

$$\forall (S, \theta_\varphi) \in \Xi, l \in L(S) \,.\, \kappa(l, \theta_\varphi) = \kappa'(l, \theta_\varphi)$$

Proof. If κ^{in} is partly labeled with respect to φ for the context pair (S', θ'_φ), then the label in the initial location $l^{\text{in}}(S')$ under context θ'_φ is already $\mathbf{t}$ or $\mathbf{f}$. Since the *labelByFixpoint* algorithm assigns labels to a location only if the location is labeled $\mathbf{m}$, it is certain that the label in $l^{\text{in}}(S')$ with respect to φ will not be changed by *labelByFixpoint*. The only way how a context pair $a = (S^a, \theta^a_\varphi)$ can influence the labels assigned to locations in another context pair $b = (S^b, \theta^b_\varphi)$ during the execution of *labelByFixpoint* is through the label in the initial location $l^{\text{in}}(S^a)$, which is used as label for calls in $C(S^b)$ which call S^a under call context θ^a_φ. Since it was shown that the label $l^{\text{in}}(S')$ in the initial location of S' is firm, the addition of (S', θ'_φ) in the labeling algorithm will not have any effect on any other context pairs $(S, \theta_\varphi) \in \Xi$. Therefore, the resulting knowledge bases κ and κ' must be equally labeled with respect to φ in all context pairs in Ξ. □

Lemma 4.5.6 (Addition of unconnected context pairs). *Let κ^{in} be a knowledge base, φ a CTL formula, and Ξ a set of context pairs obtained from the context collection and removal of partly decided contexts in Algorithm 4.1. Let $(S', \theta'_\varphi) \notin \Xi$ be a context pair which is not called from any call in any pair in Ξ:*

$$\forall (S, \theta_\varphi) \in \Xi, c \in C(S) \,.\, (S_{\leftarrow c}, \theta(c, \theta_\varphi)) \neq (S', \theta'_\varphi)$$

Let

$$\kappa \leftarrow \mathit{labelByFixpoint}(\kappa^{in}, \Xi, \varphi)$$

and

$$\kappa' \leftarrow \mathit{labelByFixpoint}(\kappa^{in}, \Xi \cup \{(S, \theta_\varphi)\}, \varphi)$$

The addition of (S', θ'_φ) to Ξ does not alter the result of labelByFixpoint with respect to all context pairs in Ξ:

$$\forall (S, \theta_\varphi) \in \Xi, l \in L(S) \,.\, \kappa(l, \theta_\varphi) = \kappa'(l, \theta_\varphi)$$

Proof. A pair which is not in the set Ξ and not called from any pair in Ξ could only affect the labels assigned to a location in a pair in Ξ by transitively affecting the label in the initial location in other pairs which are called from a pair in Ξ. Since Ξ is isolated (Lemma 4.5.4), pairs in it call only other pairs in it or partly labeled pairs. Thus, the only possibility how an unconnected pair could be connected to a pair in Ξ is via a partly labeled pair. Due to Lemma 4.5.5, the further labeling of a partly labeled pair has no effect on its callers. Therefore, the addition of an unconnected pair cannot have any effect onto pairs in Ξ. □

Proposition 4.5.7 (Collected context pairs suffice)**.** *Let κ^{in} be a knowledge base, φ a CTL formula, and Ξ a set of context pairs obtained from the context collection and removal of partly decided contexts in Algorithm 4.1. Let $(S', \theta'_\varphi) \notin \Xi$ be a context pair. Let*

$$\kappa \leftarrow \mathit{labelByFixpoint}(\kappa^{in}, \Xi, \varphi)$$

and

$$\kappa' \leftarrow \mathit{labelByFixpoint}(\kappa^{in}, \Xi \cup \{(S, \theta_\varphi)\}, \varphi)$$

The addition of (S', θ'_φ) does not alter the result of labelByFixpoint with respect to all context pairs in Ξ:

$$\forall (S, \theta_\varphi) \in \Xi, l \in L(S) \,.\, \kappa(l, \theta_\varphi) = \kappa'(l, \theta_\varphi)$$

Proof. Because of Lemma 4.5.4, a pair (S', θ'_φ) which is not in Ξ is either partly labeled or not directly called from any pair in Ξ. If the pair is partly labeled, its addition to Ξ does alter the labels assigned to locations in any other pair in Ξ, due to Lemma 4.5.5. If the pair is not directly called from any pair in Ξ, its addition to Ξ does not alter the labels assigned to locations in any other pair in Ξ, due to Lemma 4.5.6. □

Now that all differences to the basic algorithm are proven, the whole correctness can be specified:

Theorem 4.5.8 (Correctness of the Incremental Refinement)**.** *Let $\kappa = (\_ \mapsto \mathbf{m})$ be an empty knowledge base, $\mathcal{R} = (\mathbb{S}, S^{\text{in}})$ an RKS, and φ a CTL formula. A call to*

$$\kappa' \leftarrow \mathit{label}(\kappa, S^{\text{in}}, \mathit{icc}(\varphi), \mathsf{partly})$$

returns a knowledge base κ' which has the correct label with respect to the validity of φ in the initial configuration $([c^{\text{in}}], l^{\text{in}}(S^{\text{in}}))$:

$$\begin{aligned} &\kappa'(l^{\text{in}}(S^{\text{in}}), \mathit{icc}(\varphi)) = \mathbf{t} \Leftrightarrow \mathcal{R} \models \varphi \\ \wedge\, &\kappa'(l^{\text{in}}(S^{\text{in}}), \mathit{icc}(\varphi)) = \mathbf{f} \Leftrightarrow \mathcal{R} \not\models \varphi \end{aligned}$$

Proof. The correctness follows from the correctness of the basic algorithm (Theorem 3.6.1) and from the correctness of all modifications done to it (Lemma 4.5.1, 4.5.2, 4.5.3, and Proposition 4.5.7). □

Chapter 5.

Implementation

The algorithms presented in the previous chapters have been implemented by extending the model checker XMV, which is written in OCaml. The extended model checker is called RMV (recursive XMV). Because the descriptions of the approaches in the previous sections are already very precise, they allow the implementation of the algorithms without additional explanations. Therefore, this section only elaborates how the formal concepts are realized, how the data is represented, and which optimizations are made. The chapter also covers how the input for the model checker is specified and how this input is transformed into an RKS. Ultimately, a brief evaluation of the resulting implementation is conducted.

5.1. Input Specification

An important part of a model checker is the specification of its input format, that is, how sub-structures to be checked must be represented in an input file. The input format of XMV is almost equal to the one of NuSMV, which allows checking an input file with both model checkers without the necessity to rewrite it. The input format of XMV is extended to allow the specification of RKSs.

Since the input for recursive Kripke structures is based on the input for usual Kripke structures, the non-recursive part is elaborated first. Listing 5.1 shows a non-recursive input module, which displays most features of XMV. It defines a usual non-recursive Kripke structure with a left-total transition relation instead of an exit location. The syntax is equal to the one of NuSMV.

A module definition starts with the word `MODULE` followed by the name of the module. Such a module represents a Kripke structure in the non-recursive case and a sub-structure in the recursive case. The module name is especially important for recursive modules where it is used as call target for calls which call this sub-structure. The first part of a module is the definition of variable (`VAR`). Variables can either be bounded integer variables (e.g., `0, ..., 6`), enumerations (e.g., `{s1,s2,s3}`), or Boolean variables. After the variables, the labels are defined (`DEFINE`). Each label is denoted by a name and followed by a condition which defines in which states the label is present. In the example, the label `in_state2` is present in each state where the variable `state` has the value s2. The second label `cr_reset` is present in each state in which the `counter` variable is zero. The next part of a module definition (`ASSIGN`) contains the specification of the transition relation and of the initial state. For each previously defined variable, an `init` and a `next` statement has to be given. The `init` statement defines the value of the variable in the initial state. The `next` statement defines how the variable changes its values.

Listing 5.1: A Non-Recursive XMV Input Specification

```
MODULE main VAR
  counter : 0..6;
  state   : {s1,s2,s3};
  ready : boolean;
DEFINE
  in_state2 := state in {s2};
  cr_reset  := counter = 0;
ASSIGN
  init(state) := s1;
  next(state) :=
    case
      counter = 2 : {s1};
      TRUE :
        case
          state = s1 : {s2,s3};
          state = s2 : {s2};
          state = s3 : {s3,s1};
        esac;
    esac;
  init(counter) := 0;
  next(counter) := (counter + 1) mod 3;
  init(ready) := FALSE;
  next(ready) := case
    counter = 2 : !ready;
    TRUE : ready;
  esac;
SPEC AG (in_state2 -> cr_reset)
SPEC E [ in_state2 U cr_reset ]
```

The semantics of a module concerning the state space is that the value of all variables together determines a state. The initial state is the one in which all variables have the value which was given to them by the `init` statement. In the example, the initial state would be the values (0,s1,FALSE) for the variables `counter`, `state`, and `ready`, respectively. This vector notation for a state is used in the examples hereinafter. Note that the `init` statement may not yield more than one value per variable in the recursive case, since this would yield more than one initial location. For the non-recursive case, however, more than one initial state may exist.

The semantics of the `next` statement defines the transition relation. A transition from one state s1 to another state s2 exists, if the `next` statement applied to each value of s1 would possibly yield s2. As long as the `next` statement yields exactly one value for each variable, each location has only one successor location. To add transitions to more than one location, the `next` statement must yield a set value which contains more possible values. In the example, this is the case for the `state` variable. If this variable is s1, the next value is either s2 or s3. Since all other variables yield always exactly one value, states in which the variable `state` is s1 will have two successor locations. If more variables have more than one possible successor value, then all possible combinations of the values become successor states. In the example, the initial state (0,s1,FALSE) has a transition to the state (1,s2,FALSE) and to the state (1,s3,FALSE). Using this information, the Kripke structure corresponding to the input specification can be built. The Kripke structure corresponding to the specification in Listing 5.1 is shown in Figure 5.1. The upper line in each state shows the values of the variables in this state, the lower line shows labels in the state, if it has any.

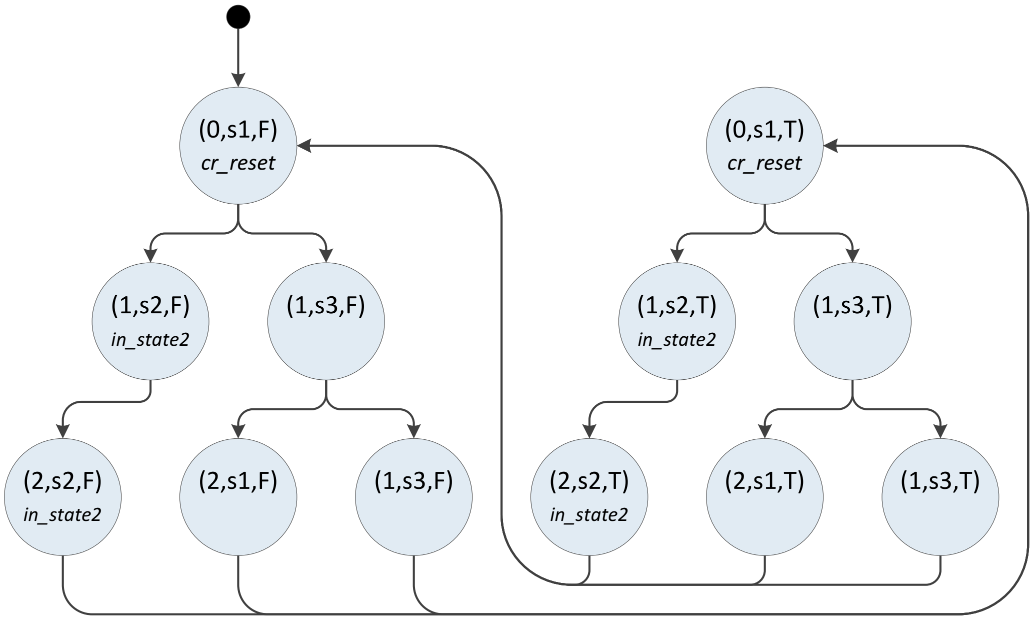


Figure 5.1.: The Kripke structure corresponding to Listing 5.1

The final part of a module are the CTL formulae which should be checked for this module. Each formula starts with the `SPEC` keyword, followed by a CTL formula which may use the labels defined in the `DEFINE` part as atomic propositions.

Recursive Modules

While a non-recursive module models a whole Kripke structure, a recursive input module instead models one sub-structure of an RKS. Therefore, multiple modules must be used to define a whole RKS. The syntax and the semantics are similar to the ones of non-recursive modules. However, the syntax is extended to allow the specification of an exit state and calls.

Listing 5.2 shows the specification of an RKS consisting of two sub-structures (modules). The differences to a non-recursive specification is the `CALL` section and the `EXIT` expression. The `CALL` section defines the calls and their targets. The syntax is similar to the one of the `DEFINE` section. However, the name in front of the assignment operator (`:=`) is no label but the name of the sub-structure which is to be called. The expression after the assignment operator is a condition which defines in which states the call is to be made. In the example, the module `foo` calls the module `bar` in the states where the variable `state` is `s1`.

The second addition is the `EXIT` expression. Each variable, even if it is of type Boolean can implicitly also have the value `EXIT`. Once a variable has this value, it keeps it forever. This means that no further `next` transitions will be considered for a variable. Once all variables have the value `EXIT`, the exit location of the sub-structure is reached. Because `EXIT` is implicitly part of the type of each variable, it does not have to be defined in the variable definition. In the example, the variable `state` in module `foo` is only defined over the enumeration `{s1,s2}`. Nevertheless, the variable can still become `EXIT`. Because the exit location is a usual location, it can have labels, like the label `q` in module `foo`, which is assigned to the exit location. The RKS represented by the input specification from Listing 5.2 is shown in Figure 5.2.

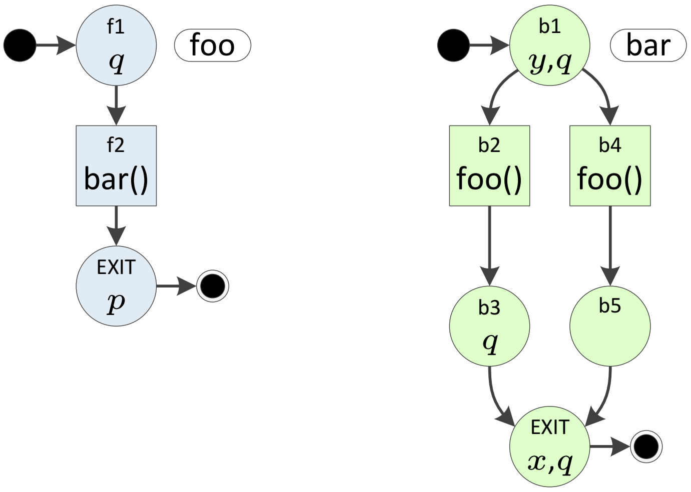


Figure 5.2.: RKS corresponding to the specification from Listing 5.2

Recursive modules share labels, that is, a label with the same name in two modules is considered to be the same label. In the example, the label `q` is treated as the same label in both module. This is necessary for specifying non-local properties. In contrast, the names of variables and enumeration constants are local. Even if variables in different

Listing 5.2: A Recursive XMV Input Specification

```
MODULE foo
VAR
  state  : {f1,f2};
DEFINE
  p := state in {EXIT};
  q := state in {f1};
CALL
  bar := state in {f2};
ASSIGN
  init(state) := f1;
  next(state) := case
    state = f1 : {f2};
    state = f2 : {EXIT};
  esac;
SPEC E [q U p]
SPEC EF q
SPEC EF p

MODULE bar
VAR
  state  : {b1,b2,b3,b4,b5};
DEFINE
  q := state in {b1,b2,EXIT};
  x := state in {EXIT};
  y := state in {b1};
CALL
  foo := state in {b2,b4};
ASSIGN
  init(state) := b1;
  next(state) := case
    state = b1 : {b2,b4};
    state = b2 : {b3};
    state = b3 : {EXIT};
    state = b4 : {b5};
    state = b5 : {EXIT};
  esac;
SPEC EF x
SPEC y
```

modules share the same name or share enumeration constants, these names are treated to be different. In the example, the variable `state` appears in both modules but with different enum constants. However, it would also be possible to share enum constants, for example, by using `s0`,...,`s1` as state constants in both modules. These variables and their values are treated as separate ones. Keeping variables and constants separate is a necessary precaution, because sharing variables would mean sharing state space which is explicitly forbidden for sub-structures of an RKS.

The final missing information is which of the sub-structures is defined to be the initial one. RMV is able to use different initial structures for different formulae. This is handy feature of RMV which allows checking different formulae for different sub-structures with one run of the program. Note that both modules have CTL formulae assigned to them. For each formula, the module in which the formula is defined is treated as initial sub-structure. For example, the formula $\mathbf{E}p\mathbf{U}q$ is defined in the module `foo`. Thus, the formula is checked using the RKS $\mathcal{R} = (\{\mathit{foo}, \mathit{bar}\}, \mathit{foo})$. In contrast, $\mathbf{EF}x$ is defined in module `bar` and therefore is checked using $\mathcal{R} = (\{\mathit{foo}, \mathit{bar}\}, \mathit{bar})$ as RKS.

RMV ensures that the modules represent well-formed sub-structures. For example, the program raises an error if a call has a label, the entry or exit location of a module is a call, or if a call is a successor of another call.

Note that the example specification in Listing 5.2 is typical for specifications for static code analysis, because it contains only one variable. Static code analysis needs to solve properties of the control flow graph. This graph is usually represented by only one variable which represents the program counter, that is, the current state in the control flow graph (it is therefore often called `state`). The state space of this variable is all the nodes of the control flow graph and the transitions are defined to reflect the edges of the graph. The states of the control flow graph have no names and are thus often only enumerated abbreviations, like the names `b1`,`b2`,... in the example.

5.2. State Space Generation

As the last section has shown, the input specification has to be transformed to a corresponding RKS (or merely a Kripke structure in the non-recusrive case) before the model checking can be conducted. Since RMV represents the state space explicitly, the state space of an RKS can be built by graph exploration. The exploration starts at the initial location, which can be deduced by evaluating all `init` expressions. Starting from this location, the outgoing transitions can be computed by evaluating the `next` expression. For each variable, the corresponding expression yields a set of values which the variable can have in the next step. The cross product of the set for each variable yields the successor states. For example, if a sub-structure consists of three integer variables a, b, and c and the evaluation of the next expressions of these variables yield the values $A = \mathit{next}(a) = \{1, 2\}$, $B = \mathit{next}(b) = \{5, 6\}$, and $C = \mathit{next}(c) = \{8, 9\}$, then the resulting set of successor states is

$$A \times B \times C = \{\{1,5,8\},\{2,5,8\},\{1,6,8\},\{1,5,9\},\{2,6,8\},\{2,5,9\},\{1,6,9\},\{2,6,9\}\}$$

After the successor states have been computed, they are visited recursively. Each visited state is created, saved, and the transition to that state which led to it is created. A state which has already been created is not visited again, but a transition is created to it whenever it appears again as a successor. After the recursive visiting is finished, the complete state space is created.

After the state space is built, the labels are placed. This is done by evaluating the expression for each label definition. The resulting set of states is then assigned the corresponding label.

The state space and label generation is only a preprocessing step which is performed before the actual model checking. Therefore, it should not take more time than the model checking itself. To allow the state space and label generation in a reasonable period of time, some optimizations have to be conducted. The computation of the `next`-expression for a variable can be non-trivial. It is often a `case` construct which performs a case distinction over the value of the variable. Thus, the amount of cases is equal to the amount of values the variable can have. If a case statement is evaluated naïvely, that is, by evaluating each case expression sequentially and stopping once the first expression evaluates to true, the process of finding the correct case is in $O(n)$ where n is the number of cases. Since this computation has to be done again in each state, the overall complexity is $O(n \times m)$ where m is the number of states. If the state space of a sub-structure is represented by a single variable, then the number of cases is usually equal to the number of states, yielding $O(n^2)$. It is obvious that this complexity can become a problem even for moderately large sub-structures. Therefore, it is viable to check whether a case statement can be evaluated using a table lookup. If all expressions are of the form $x = c$ where x is the next-variable and c is one of its possible values, then it is possible to evaluate the `case` construct using a lookup table, thus requireing only $O(1)$, yielding $O(n)$ for the state generation process.

Another important optimization is to introduce some state indexing to allow the faster assignment of labels. Naïvely, the set of states which is to be labeled with a label is obtained by checking for each state whether the expression that defines the label holds in that state, yielding a complexity of $O(s \times l)$ where s is the number of states and l is the number of labels. When having many states and labels, this again yields a quadratic complexity. By keeping an index, which, for example, orders all states with respect to the value of one variable, the complexity can be reduced for simple expressions like `state = s1` by looking up only states for which the variable `state` is `s1` instead of testing all states.

Even more optimizations like constant expression propagation were performed in the implementation. Without optimizations, the time for building and labeling the state space of larger sub-structures was often longer than the time for model checking these sub-structures. With all optimizations, the time is now usually neglectable compared to the model checking time. Note that all optimizations presented here also apply to the non-recursive part of RMV. Here, they yielded a large performance gain, as well.

5.3. Data Representation

RMV is an explicit model checker. Therefore, the state space of the sub-structures (but of course not of the possibly infinite semantic Kripke structure) is built and represented explicitly. Since most of the algorithm works on this explicit state space, its representation is very important for the performance of the algorithm.

The first decision concerning state space representation is the representation of locations and calls, which are hereinafter also referred to as *states*. Using a good representation speeds up other data structures which operate on the states, like the labeling function μ and the call target function ν.

An easy representation, which is also very beneficial for mappings, is to use integers to represent states of a sub-structure. These integers should be very dense to allow using them as indexes for densely packed arrays. For example, a sub-structure with ten states should represent them with the number $0, \ldots, 9$. Such dense integers can be obtained by performing a traversal of the sub-structure and enumerating the states during the traversal, starting with zero for the initial state.

While a hash or tree map would be the most natural implementation for mappings like μ and ν, the dense numbering of states allows to use ordinary arrays for these mappings. The numbers of the states can directly be used as indexes in these arrays. This is faster than using hashes and also takes up less space. In addition, the data is packed more densely than a hash map which also promises better caching behaviour.

In addition to the states, also the labels are also represented by integers by enumerating all labels in a sub-structure. Thus, the labeling function μ can be implemented as an array of integer sets. The representation of the integer sets can range from a tree set, to a list, or even a bit-set. In the case of RMV, tree sets were used. The transition relation δ can be expressed in the same way. This allows fast look-up of the successors of a state, which is directly given by the set found in the array at the index of the state. Since the model checker also needs to look-up the predecessors of a state, a reverse transition mapping δ^{-1} is maintained, as well. The call target mapping ν can be implemented as an array of pointers to other sub-structures.

Because the initial location always receives the index 0, it does not have to be represented explicitly. The set of states $L \cup C$ is simply expressed by the number of states n, which represents the set of states $\{0, \ldots, n-1\}$.

In addition to all these components, a sub-structure implementation must also contain mappings from the indexes of states and labels back to their respective string representation. These mappings are needed for the visual output of the program which should for example state the label names instead of their indexes. Since these mappings are not needed for the model checking itself, their performance is only of a very minor impact. Therefore, hash maps were used for these purposes.

A sub-structure is represented as a struct of the aforementioned components. A recursive Kripke structure is implemented as a list of sub-structures. The initial sub-structure is represented implicitly by the convention that the first sub-structure in the list is the initial one.

Array Versus Struct Representation

The current implementation uses only integers as states and represents all further functions, relations, and mappings as arrays. Another approach with high performance could be to model each state as a struct which then for instance has its successors states and its labels as fields. This would allow even faster access than the current approach which always needs an array look-up. However, the array approach has significant advantages:

- The caching behaviour is better since all data is kept close together in an array.

- The basic model checking process requires many set operations on sets of states like union and intersection. These operations would be complex and costly if each state was represented as a struct. By using arrays, states are only integers, and can be stored conveniently in tree sets or bit sets which offer fast implementations for set operations. For example, a bit set can use the very fast bitwise-or operation as implementation for the set union operation.

- The algorithm often needs to create and initialize a sub-structure from another one and change for instance only the labeling ν. Using arrays, it is easy to replace the labeling ν while reusing all other components of the old sub-structure. This can be done by having the new sub-structure point to the old arrays and only use a new array for the labels. If states were represented as structs, it would be necessary to replicate each state in a sub-structure to create a new sub-structure which only differs in its labeling. This would yield a significant slowdown of the algorithm and increase the memory consumption.

These disadvantages of the struct representation leaded to the decision to use the array representation for the implementation of RMV.

5.4. Optimizations

There are some optimizations of the algorithms presented in this thesis which were omitted until now, because they are not conceptually useful. However, such optimizations yield a noticeable speed up and therefore should be included in an implementation which is to be used for large inputs. Therefore, they are shortly presented here.

The most important optimization is a more sophisticated implementation of the fixed point algorithm. As depicted in the previous chapters, the fixed point algorithm naïvely iterates over all sub-structures and tries to deduce new labels using local model checking of the associated Kripke structure. Since the number of iterations necessary growths with the formula depth, this can lead to a number of iterations where *each* sub-structure is checked again in each iteration although no new results for that sub-structure can be deduced. Consider the example from page 58 where an RKS is checked against a nested **EF** formula.

Table 5.1.: Fixed point algorithm results from previous example

Context	fff	fft	ftt	tff	tft	ttt
c1	$\mathbf{t}(4)$	$\mathbf{t}(2)$	$\mathbf{t}(2)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$
c3	$\mathbf{f}(1)$	$\mathbf{f}(1)$	$\mathbf{f}(1)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$	$\mathbf{t}(1)$
d1	$\mathbf{t}(3)$	-	$\mathbf{t}(1)$	$\mathbf{t}(1)$	-	$\mathbf{t}(1)$
d3	$\mathbf{f}(1)$	-	$\mathbf{t}(1)$	$\mathbf{t}(1)$	-	$\mathbf{t}(1)$

Table 5.1 again depicts the result of the fixed point algorithm for formula $\mathbf{E}\top\mathbf{U}(p \wedge \mathbf{E}\top\mathbf{U}(q \wedge \mathbf{E}\top\mathbf{U}p))$ in the example. As shown in the table, the fixed point algorithm needs four iterations. In each iteration, all ten context pairs are model checked locally, yielding 40 local checks. The table however shows that it would be sufficient to check all ten context

pairs only once and then only $(C, \mathbf{fft})$ and $(C, \mathbf{ftt})$ in the second step, $(D, \mathbf{fff})$ in the third step and $(C, \mathbf{fff})$ in the fourth, because only these context pairs change in the respective steps. Performing such checking would yield only $10 + 2 + 1 + 1 = 14$ instead of 40 local checks, which is a very significant speed-up. The contexts which have to be checked after a step can easily be inferred if back-pointers from a context pair to all pairs which call that pair are kept: Whenever the validity of the initial location of a sub-structure in a specific context pair x is switched from $\mathbf{m}$ to $\mathbf{t}$ or $\mathbf{f}$, then only context pairs calling x have to be re-checked instead of *all* context pairs.

Another optimization is the omitting of context assumptions for atomic propositions, because these are only important for **EX** formulae. For all other formulae, the keeping of these context assumptions increases the number of reachable context pairs unnecessarily. For example, the formula $\mathbf{E}p\mathbf{U}q$ has only two possible non-$\mathbf{m}$ contexts (namely $\langle\mathbf{f}\rangle\mathbf{E}p\mathbf{U}q$ and $\langle\mathbf{t}\rangle\mathbf{E}p\mathbf{U}q$) when not considering assumptions about atomic propositions but eight possible contexts with these assumptions. Either **EX** could be left out completely, since it is of no use in static analysis anyway, or an approach comparable to the one of Fehnker et al. (cf. Appendix A) can be used to solve **EX**. The context assumptions for atomic propositions were only presented here for simplicity reasons, because they allow to solve **EX** in a very natural, easily provable way.

The next point which was left unoptimized due to simplicity reasons is the context collection. The *collectContext* operation always collects both versions of a call context, one with $\mathbf{t}$ as topmost context assumption and one with $\mathbf{f}$ as topmost context assumption. The reason for this is that the topmost context is not known yet and keeping both contexts ensures that always the correct one is included. There are, however, some formulae for which the correct context can be inferred easily. An example for this would be $\varphi \equiv \neg\psi$. For this formula, the topmost context assumption is always the opposite of the context assumption of ψ. All non-temporal operators allow to infer the context assumption that easily. By collecting only the correct context whenever it can be inferred directly, the number of contexts to be model checked is reduced.

A final optimization which is yet to be introduced to the implementation of RMV does not concern the algorithm itself, but the data representation. Currently, the model checking process uses the basic OCaml implementation of tree-sets for representing sets of states. The advantage of these sets, when, for example, compared to hash-sets is that they allow the executions of frequently needed set operations like union, difference and intersection very efficiently. A better representation, which can efficiently be executed on modern hardware would be bit sets. These sets ususally consume less memory and the set operations can be implemented very efficiently using bitwise operations. Since modern hardware with vector extensions allows to perform these operations on more than one integer concurrently, they can be even more efficient.

5.5. Evaluation

Because Goanna is currently not yet able to produce RKS specifications from C/C++ code but only "usual" non-recursive Kripke structure specifications, the performance of the algorithm cannot be evaluated using real static analysis examples, yet.

Concerning the non-recursive model checking algorithm, XMV, which was also optimized during this thesis, is now very well suited for static code which usually consists of state spaces with less than a million states but many formulae to be checked. As an example, a randomly generated single-module input specification with 5000 states and 4500 formula specifications was model checked on a modern laptop in 0.372 seconds by XMV while the latest release of NuSMV needed 9.57 seconds. This landslide victory for XMV shows that explicit state representation is the representation of choice for the kinds of problems for which XMV is designed.

Chapter 6.

Conclusion & Future Work

In this thesis, an algorithm for model checking of recursive Kripke structures and an incremental refinement of that algorithm were proposed. It was reasoned about the formal basis for these algorithms and their correctness was proven. The algorithms were also implemented into the model checker XMV.

The thesis started with the formal definition of Kripke structures, CTL and its semantics on Kripke structures, and finally recursive Kripke structures including execution and CTL semantics. Then, a method for reducing the possibly infinite number of reachable call stacks to a finite number of equivalence classes was proposed. It was argued that two call stacks calling the same sub-structure S are equal with respect to the model checking of a CTL formula φ, if the same subformulae of φ, including φ itself, are true in configurations which are reached once S exits. This set of valid subformulae, which was presented in a tree-like manner, was coined call context. Next, the knowledge base κ was introduced which stores the already deduced validity values of formulae in certain configurations. Afterwards, an algorithm for model checking RKSs with CTL, which relies on the introduced concepts, is presented and its correctness is proven. Subsequently, a refinement of the algorithm is proposed which performs only local model checking and reduces the problem complexity by checking formulae incrementally only in sub-structures which are necessary to yield a correct result in the initial configuration. In the next chapter, the implementation of the algorithms into the model checker XMV (yielding the model checker RMV) was depicted. Here, the structure of the input specification was shown, appropriate data structures and mechanisms for optimizing the algorithm and yielding a high performance were discussed, and a brief evaluation of RMV was conducted.

An obvious topic for future work is a comprehensive evaluation of the algorithms by benchmarking their performance for static analysis of large software projects, like, for example, Firefox or the Linux kernel. Such study could yield how large the performance increase of the incremental refinement, when model checking RKSs consisting of thousands of sub-structures, is. A related topic would be the benchmarking against other static analysis tools which use other mechanisms like pushdown automata for the model checking.

The current approach is limited to labels only based on the location. An interesting topic would be to increase the expressiveness of the model by allowing certain properties with respect to stack content, scope, or call hierarchy. To keep the applicability for static analysis of large projects, the additional features must be chosen carefully. Otherwise, the resulting problem could be to hard to solve with the required performance or even be undecidable.

Another possibility for increasing the expressiveness of the model would be the introduction of local, parametrized labels. Currently, all labels are global to the whole RKS.

This is appropriate when checking, for example, properties of global variables, like checking whether a global or local pointer variable is dereferenced when it is null. These labels are ,however, not fully appropriate for checking the same property for pointer variables which are handed as parameters to a function f. In this case, reads and writes to that parameter could be encoded as parametrized labels. For example, a label $\mathsf{read}(x)$ could be used whenever a parameter x is read. When the function (i.e., the sub-structure) f is called with two different variables a and b as argument, the parameter can be substituted with the argument thus forming the labels $\mathsf{read}(a)$ and $\mathsf{read}(b)$. When considering a formula with parametrized labels like $\mathbf{EG}\,\mathsf{read}(x)$, the algorithm would model check f only once with respect to $\mathsf{read}(x)$ and then use the results for model checking of functions that call f for both formulae $\mathbf{EG}\,\mathsf{read}(a)$ and $\mathbf{EG}\,\mathsf{read}(b)$, depending on the variable which is used in the call to f. In addition, this approach could allow to define wildcard formulae like $\mathbf{AF}\,\mathsf{write}(*)$, which encode that for each variable, there is a path where it is written. It is to be investigated how this can be handled without explosion of the model checking complexity. In addition, it must be checked how the model must be extended to allow these parametrized labels.

In conclusion, there is still a lot of work to be done in the field of model checking recursive Kripke structures and thus a lot of possibilities for future research in this field exist.

Appendix A.

Basic Approach

This chapter depicts the algorithm of Fehnker et al. [44] on which the algorithms in this thesis are based. In addition to renarrating the concepts of their algorithm, the parts of the algorithm which were not described formally in [44] are formalized here. Finally, a proof sketch for the correctness of the algorithm is given.

A.1. Overview

Because the algorithm elaborated in this thesis is based on the basic algorithm presented in this chapter, many parts of the basic algorithm are similar to the algorithm presented in Chapter 3. As already mentioned there, the biggest difference is that the basic algorithm assembles a new RKS $\mathcal{R}'$ after checking a formula φ. This RKS is used as input for checking formula which include φ as subformula. The validity of φ is inserted as labels in $\mathcal{R}'$ and a sub-structure S which is called in different contexts is represented by copies of that sub-structure, where each copy represents one contexts. Calls to S are redirected to the correspondent copy by choosing the respective copy based on the context of that call.

The model checking of a formula φ is done via summaries which aggregate the knowledge about the validity of φ in locations and calls. The goal in each step is to label all locations in a summary. Then, a new RKS is built from the summary.

The algorithm presented here differs slightly from the algorithm presented in [44]. First, other notations and namings are used. For example, boxes are called calls here and the external assumption is called context assumption. Next, an RKS is no longer a sequence of sub-structures but a set of sub-structures with a dedicated initial structure. As a contribution of this thesis, the algorithms *split* and *merge* were formalized here. The final difference is that a distinction between ternary values in the summary and labels in the associated Kripke structure is made. In [44], an assumption $\mathbf{t}$ led to a $\mathbf{t}$ label in the associated Kripke structure. This was confusing, because it was not obvious that $\mathbf{t}$ is only an atomic proposition in the Kripke structure instead of the formula $\top$. To avoid this possible confusion special $p^{\mathbf{t}}$ and $p^{\mathbf{m}}$ labels are used in the associated Kripke structure for the assumptions $\mathbf{t}$ and $\mathbf{m}$, respectively.

A.2. Summary Generation

This section depicts how the basic approach generates fully labeled summaries for a formula φ, given an RKS which is already labeled with respect to subformulae of φ, which means that its locations l are labeled with atomic propositions ψ where the subformula ψ holds.

A.2.1. Local Checking & Context Assumptions

The core idea of the algorithm is to reduce the global problem of solving a formula for the whole RKS to a local problem which solves a formula for only a sub-structure disregarding all other structures, despite some assumptions about sub-structures which are called by the currently considered sub-structure.

The local checking is conducted by building an associated Kripke structure from the sub-structure to be checked. A labeling is chosen for the built Kripke structure which reflects the local labeling of the sub-structure and the assumptions about sub-structures which are called by calls in the sub-structure. By choosing an appropriate labeling for the Kripke structure, it combines the labeling of the sub-structure with the assumptions about calls in that structure. The constructed Kripke structure is then checked by a usual model checker for Kripke structures.

The problem with the local solution of a formula for a sub-structure is that its outcome might depend on the call context, that is, on the sub-structure which calls the one under consideration. Assumptions have to be made about the successor locations of the call in the calling structure. To be precise, it has to be assumed whether the formula under consideration holds in the successor locations. Since these successor locations are in the call context of the sub-structure under consideration, the assumption about them is called *context assumption* $\eta \in \mathbb{T}$. By model checking a sub-structure with $\eta = \mathbf{t}$ and $\eta = \mathbf{f}$, the value for $\eta = \mathbf{m}$ can be derived using the $\bowtie$ operator.

A.2.2. Summaries

Definition A.2.1 (Summaries). *Let $\mathcal{R} = (\mathbb{S}, S^{\text{in}})$ be an RKS and $S = (L, l^{\text{in}}, l^{\text{out}}, C, \nu, \delta, \mu)$ a sub-structure of $\mathcal{R}$. A* local summary *γ_S for S is a pair (α, ω), where $\alpha : \mathbb{T} \times C \Rightarrow \mathbb{T}$ is an* assumption function *and $\omega : \mathbb{T} \times L \Rightarrow \mathbb{T}$ is a* guarantee function, *such that $\alpha(\mathbf{m}, c) = \alpha(\mathbf{t}, c) \bowtie \alpha(\mathbf{f}, c)$ for all $c \in C$ and $\omega(\mathbf{m}, l) = \omega(\mathbf{t}, l) \bowtie \omega(\mathbf{f}, l)$ for all $l \in L$. The class of all local summaries is depicted by Γ.*

A global summary *$\pi : \mathbb{S} \to \Gamma$ for $\mathcal{R}$ is a mapping which assigns a local summary γ to each sub-structure of $\mathcal{R}$, that is, $\pi(S) = \gamma_S$ for all $S \in \mathbb{S}$. The class of all global summaries is depicted by Π.*

The term summary *is used for both — local and global summaries — if it can be deduced from the context which one is meant.*

A summary depicts knowledge about the validity of a formula. The terms γ^{φ} and π^{φ} are used to denote that the respective summary depicts information about formula φ.

For a sub-structure S and a CTL formula φ, the summary $\gamma = (\alpha, \omega)$ for that sub-structure with respect to φ contains information about the validity of φ in calls and locations of S. The assumption function $\alpha(\eta, c)$ states which assumption is made about call $c \in C(S)$, if S is called in a context with context assumption $\eta \in \mathbb{T}$. If the result is $\mathbf{t}$ or $\mathbf{f}$, then the summary already assumes that φ holds or does not hold in c, respectively. In contrast, if the result is $\mathbf{m}$, then no assumption can be made about the call c with respect to

φ, yet. The guarantee function $\omega(\eta, l)$ states which guarantees can already be made about locations $l \in L(S)$, if S is called in a context with context assumption $\eta \in \mathbb{T}$. If the result is $\mathbf{t}$ or $\mathbf{f}$, then φ is guaranteed to hold or not to hold in l, respectively. If the result is $\mathbf{m}$, then no guarantee can be given for l, yet.

Generally, the value $\mathbf{m}$ in the guarantee or assumption function represents that either the sub-structure has not been checked yet or that the result was inconclusive. The latter might be the case if the result depends on other unchecked or inconclusive summaries.

The context assumption η used as first parameter in the guarantee function ω and assumption function α of a summary γ^φ for sub-structure S with respect to formula φ is equal to the topmost context assumption η in the call context $\theta_\varphi = \langle\eta\rangle\tau$ in the approach of this thesis: It states whether φ holds in the successor location $l^{\text{succ}}(c)$ of the call c which calls the sub-structure S. For example, a concrete assumption value $\alpha(\mathbf{t}, c) = \mathbf{t}$ can be interpreted as "If it is assumed that φ holds in the successor location of a call which calls S, then it can be assumed that φ holds in the initial location $l^{\text{in}}(S_{\leftarrow c})$ of sub-structure $S_{\leftarrow c}$ which is called by c. Note that only the values for the context assumptions $\mathbf{t}$ and $\mathbf{f}$ have to be maintained explicitly for the guarantee and assumption functions. The values for the $\mathbf{m}$ context assumptions can be derived with the $\bowtie$ operator. Similarly to the knowledge base, a summary can also be insufficiently labeled, partly labeled or fully labeled:

Definition A.2.2 (Labeled Summary). *A summary $\gamma = (\alpha, \omega)$ for a sub-structure $S = (L, l^{\text{in}}, l^{\text{out}}, C, \nu, \delta, \mu)$ is called* partly labeled*, if its guarantee function is $\mathbf{t}$ or $\mathbf{f}$ in the initial location l^{in} for both context assumptions $\mathbf{t}$ or $\mathbf{f}$:*

$$\forall \eta \in \{\mathbf{t}, \mathbf{f}\} \,.\, \omega(\eta, l^{\text{in}}) \in \{\mathbf{t}, \mathbf{f}\}$$

A summary is called fully labeled*, if its guarantee function is $\mathbf{t}$ or $\mathbf{f}$ in all locations $l \in L$ for both context assumptions $\mathbf{t}$ or $\mathbf{f}$:*

$$\forall l \in L, \eta \in \{\mathbf{t}, \mathbf{f}\} \,.\, \omega(\eta, l) \in \{\mathbf{t}, \mathbf{f}\}$$

A summary is called unlabeled *or* insufficiently labeled *if it is not partly labeled.*

A partly labeled summary has already its initial location decided. Thus, it is labeled with respect to its callers. This means that assumptions other summaries make about calls which call the sub-structure of this summary can be updated to the value of the initial location l^{in} and will not be subject to change afterwards.

A fully labeled summary has all its locations labeled. It is completely model checked with respect to φ. Note that a fully labeled summary is also partly labeled.

An unlabeled summary hasn't got its initial location labeled for all context assumptions. It may have other labeled locations and the initial location may be labeled for one - but not all - context assumptions.

A.2.3. Context Assumptions

The basic approach does not restrict a call to have only one successor location. Therefore, the context assumption of a call with respect to a formula φ is no longer only the validity of φ in the successor location, but the logical disjunction of the validity of φ in all successor locations. This fact is expressed by the context assumption function $ca : \Gamma \times C \times \mathbb{T} \rightarrow \mathbb{T}$, which looks up the context assumption of a call. It is defined as follows:

$$ca((\alpha, \omega), c, \eta) = \bigvee_{l \in L(S(c))\,.\,(c,l) \in \delta_{S(c)}} \omega(\eta, l) \tag{A.1}$$

The function looks up the guarantee for all successor locations of call c (in a given summary γ and under a given context assumption η) and combines them by logical disjunction. For example, if sub-structure S_1 calls S_2 in call c (i.e., $\nu_{S_1}(c) = S_2$) and the context assumption for c under a given summary γ_{S_1} and a given context assumption η is $\mathbf{t}$, then S_2 has to be checked with the context assumption $\mathbf{t}$.

It is sufficient to only use successor locations and not successor calls in the definition of *ca*, because the definition of a recursive Kripke structure ensures that a call may never be followed directly by another one. The use of logical disjunction for the determination of the context assumption of a called sub-structure is justified because the algorithms use the orthogonal CTL operators which only consists of $\mathbf{EX}\psi$, $\mathbf{EG}\psi$, and $\mathbf{E}\psi\mathbf{U}\chi$. All these operators use an existential quantification. Therefore, if the formula holds in at least one successor location, then $\mathbf{t}$ is the correct context assumption. This semantics is satisfied by combining the values by logical disjunction.

A.2.4. Sub-structure to Kripke Structure Transformation

The basic approach also constructs an associated Kripke structure from a sub-structure to infer labels for a CTL formula φ by local model checking. The associated Kripke structure $\mathcal{K}$ has exactly the same structure as in Chapter 3. However, instead of inferring the labels for subformulae of φ from the knowledge base, the labels for these subformulae are already represented as atomic propositions in the RKS. Therefore, it suffices to use these labels in $\mathcal{K}$. The labels for the validity of φ in calls and in the state ω are taken from the assumption function α of the summary and from the context assumption η, respectively. The formal definition of $\mathcal{K}$ for the basic approach is shown in Definition A.2.3.

Definition A.2.3 (Associated Kripke Structure)**.** *Let S be a sub-structure, $\gamma = (\alpha, \omega)$ be a summary for S, $\varphi \equiv \mathbf{EG}\psi \mid \mathbf{E}\psi\mathbf{U}\chi$ a CTL formula, and $\eta \in \mathbb{T}$ be a context assumption. Let $AP = \{\ulcorner\psi\urcorner, \ulcorner\chi\urcorner, p^{\mathbf{t}}, p^{\mathbf{m}}\}$ be a set of atomic propositions. Let $\alpha : \mathbb{T} \times AP \rightarrow \wp(AP)$ be a function which is defined as follows:*

$$\alpha(t, p) = \begin{cases} \{p\} & \text{if } t = \mathbf{t} \\ \emptyset & \text{otherwise} \end{cases}$$

Let $\beta : \mathbb{T} \rightarrow \wp(AP)$ be a function which is defined as follows:

$$\beta(\eta) = \begin{cases} \{p^{\mathbf{t}}\} & \text{if } \eta = \mathbf{t} \\ \{p^{\mathbf{m}}\} & \text{if } \eta = \mathbf{m} \\ \emptyset & \text{otherwise} \end{cases}$$

The associated Kripke structure *$\mathcal{K}(S, \alpha, \eta, \varphi)$ is a transition system $(\mathcal{S}, I, \delta, \mu)$ over the set of atomic propositions AP with*

- *the set of states $\mathcal{S} = L \cup C \cup \{\omega\}$*

- *the set of initial states* $I = \{l^{\text{in}}\}$
- *the transition relation* $\delta = \delta \cup \{(l^{\text{out}}, \omega), (\omega, \omega)\}$
- *the labelling function* μ *which is defined as follows:*

$$\mu(l) = \begin{cases} \alpha(\psi \in \mu_S(l), \ulcorner\psi\urcorner) & \text{if } \varphi \equiv \mathbf{EG}\psi \\ \alpha(\psi \in \mu_S(l), \ulcorner\psi\urcorner) \cup \alpha(\chi \in \mu_S(l), \ulcorner\chi\urcorner) & \text{if } \varphi \equiv \mathbf{E}\psi\mathbf{U}\chi \end{cases} \quad \text{for all } l \in L$$

$$\mu(c) = \beta(\alpha(\eta, c)) \qquad \text{for all } c \in C$$

$$\mu(\omega) = \beta(\eta)$$

While the new approach built an associated Kripke structure $\mathcal{K}(\kappa, S, \theta_\varphi)$ from a knowledge base, a sub-structure, and a context, the basic approach builds $\mathcal{K}(S, \alpha, \eta, \varphi)$ from a sub-structure an assumption function, a context assumption, and the formula to be checked. The assumption function is equal to the knowledge base: Assumptions for calls are retrieved from it. The context assumption η is equal to the topmost context assumption of θ_φ. Thus, the only real difference is that location labels are taken directly from the sub-structure labeling μ instead of out of the knowledge base. As already mentioned, this requires that all subformulae of φ are present as atomic propositions in AP.

Another difference to $\mathcal{K}$ in the new approach is that the call and ω state labels are not called $\ulcorner\varphi\urcorner$ and $\ulcorner\varphi_?\urcorner$ but rather $p^{\mathbf{t}}$ and $p^{\mathbf{m}}$, respectively. This is the case, because these labels do not necessarily express the validity of φ. Based on the structure of φ, they either represent the validity of φ or in case of $\varphi \equiv \mathbf{EX}\psi$, they represent the validity of ψ.

A.2.5. Guarantee Coherence

The local coherence of a summary describes whether the guarantees can be deduced from the current assumptions (i.e., whether it can be proven by model checking that the guarantees hold). Incoherent guarantees which cannot be deduced, are hazardous because they might turn out to be wrong. Continuing the model checking with incoherent guarantees might lead to an overall wrong result. Therefore, the algorithm may never yield incoherent guarantees. Definition A.2.4 shows the formal specification of coherence and maximal coherence.

Definition A.2.4 (Locally Coherent Summary). *Given an RKS* $\mathcal{R} = (\mathbb{S}, S^{\text{in}})$ *over* $AP \supseteq \{\psi, \chi\}$*, a sub-structure* $S = (L, l^{\text{in}}, l^{\text{out}}, C, \nu, \delta, \mu)$ *of* $\mathcal{R}$*, and a CTL formula* $\varphi \equiv \mathbf{EX}\psi \mid \mathbf{EG}\psi \mid \mathbf{E}\psi\mathbf{U}\chi$ *. Let* $\gamma = (\alpha, \omega)$ *be a summary for* S *with respect to* φ*. Let* $\varphi_p = \varphi\{\psi \mapsto \ulcorner\psi\urcorner, \chi \mapsto \ulcorner\chi\urcorner\}$ *be a CTL formula in which direct subformulae* ψ *and* χ *of* φ *are replaced by atomic proposition* $\ulcorner\psi\urcorner$ *and* $\ulcorner\chi\urcorner$*. A guarantee* $\omega(\eta, l)$ *for a location* $l \in L$ *and a context assumption* $\eta \in \mathbb{T}$ *is called* coherent*, if it satisfies:*

Case $\varphi \equiv \mathbf{EX}\psi$

1. $\omega(\eta, l) = \mathbf{t} \;\Rightarrow\; \mathcal{K}(S, \alpha, \eta, \varphi), l \models \mathbf{EX}(\ulcorner\psi\urcorner \vee p^{\mathbf{t}})$
2. $\omega(\eta, l) = \mathbf{f} \;\Rightarrow\; \mathcal{K}(S, \alpha, \eta, \varphi), l \models \neg\mathbf{EX}(\ulcorner\psi\urcorner \vee p^{\mathbf{t}})$

Case $\varphi \equiv \mathbf{EG}\psi \mid \mathbf{E}\psi\mathbf{U}\chi$

1. $\omega(\eta, l) = \mathbf{t} \;\Rightarrow\; \mathcal{K}(S, \alpha, \eta, \varphi), l \models \quad \varphi_p \;\vee \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}p^{\mathbf{t}}$
2. $\omega(\eta, l) = \mathbf{f} \;\Rightarrow\; \mathcal{K}(S, \alpha, \eta, \varphi), l \models \neg(\varphi_p \;\vee\; \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}p^{\mathbf{t}} \;\vee\; \mathbf{E}\ulcorner\psi\urcorner\mathbf{U}p^{\mathbf{m}})$

The summary γ is called coherent, *if $\omega(\eta, l)$ is coherent for all $\eta \in \mathbb{T}$ and all $l \in L$. The summary π for $\mathcal{R}$ is called* coherent, *if $\pi(S)$ is coherent for all $S \in \mathbb{S}$. A summary γ or π is called* maximally coherent, *if the equations above hold with the implications ($\Rightarrow$) replaced by equivalences ($\Leftrightarrow$).*

The equations in the definition of coherence are the basis for the *deduceLabels* function in the new approach (cf. Definition 3.5 on page 50). The difference is only that the function in the new approach is only used to label **EG** and **EU** formulae while the basic approach also labels **EX** using the definition of coherence. Because of that, the definition of the associated Kripke structure must use the $p^{\mathbf{t}}$ values instead of $\ulcorner\varphi\urcorner$ for calls and the ω state. In the case of $\varphi \equiv \mathbf{EX}\psi$, the validity of φ in the initial location of a sub-structure S does not imply the validity in locations which are predecessors of a call which calls S. Instead, the validity of ψ in the initial location does imply the validity of φ for predecessor locations of a call to S. Because of that, the $p^{\mathbf{t}}$ represents the validity of ψ in this case.

A value of $\mathbf{m}$ is always coherent, as it states no decision and therefore is never incorrect. For $\mathbf{t}$ and $\mathbf{f}$, the respective implication must hold for a coherent guarantee. Of course, the implications for $\mathbf{t}$ and $\mathbf{f}$ are mutually exclusive. Note that coherence is only defined for guarantees and thus locations. Coherence puts no restriction on assumptions made about calls. However, the assumptions influence the coherence of guarantees, of course.

A summary is called maximally coherent if it is not possible to change a guarantee from $\mathbf{m}$ to $\mathbf{t}$ or $\mathbf{f}$ without making it incoherent. Informally, maximal coherence states that all guarantees which *can* be decided *are* decided. For example, in case of $\varphi = \mathbf{EX}p$, if $\mathcal{K}(S, \alpha_S, \eta, \varphi), l \models \mathbf{EX}(\ulcorner\psi\urcorner \vee p^{\mathbf{t}})$, then $\omega_S(\eta, l)$ must be $\mathbf{t}$ to be maximally coherent. If it was $\mathbf{m}$, then the summary would not maximally coherent, but still coherent. If it was $\mathbf{f}$, then the summary would be incoherent.

The algorithm uses the definition for maximal coherence to make a summary maximally coherent. This is done by switching a guarantee for a location l from $\mathbf{m}$ to $\mathbf{t}$ or $\mathbf{f}$ when the respective formula from the coherence definition holds in l. Like in the new approach, the validity of the formulae (i.e., whether $\mathcal{K}(S, \alpha_S, \eta, \varphi), l \models \psi$) can be computed by model checking of the (non-recursive) associated Kripke structure.

A.2.6. Assumption Consistency

In contrast to the new algorithm, the basic algorithm saves assumptions made about calls explicitly in the assumption function α of a summary. These assumptions have to be updated after guarantees change to reflect the latest guarantees for initial locations. Assumptions that reflect the latest guarantees are called *consistent*. A summary which only contains consistent assumptions is also called *consistent*.

Consistency of assumptions is the counterpart to coherence of guarantees. While coherence is used to decide guarantees, consistency is used to decide assumptions. During the checking process, more and more sub-structures become partly labeled (i.e., their initial location gets labeled). The assumptions for calls which call these sub-structures can then be switched from $\mathbf{m}$ to either $\mathbf{t}$ or $\mathbf{f}$. Definition A.2.5 depicts the formal specification of consistency.

Definition A.2.5 (Consistency). *Let $\varphi \equiv \mathbf{EX}\psi \mid \mathbf{EG}\psi \mid \mathbf{E}\psi\mathbf{U}\chi$ be a CTL formula and let $S = (L, l^{\mathrm{in}}, l^{\mathrm{out}}, C, \nu, \delta, \mu)$ be a sub-structure of an RKS $\mathcal{R} = (\mathbb{S}, S^{\mathrm{in}})$ over atomic propositions $AP \supseteq \{\psi, \chi\}$. Let $\gamma = (\alpha, \omega)$ be a summary for a sub-structure S with respect to φ and $\gamma_c = (\alpha_c, \omega_c)$ be a summary with respect to φ for a sub-structure $S_{\leftarrow c}$ which is called by a call $c \in C$. An assumption $\alpha(\eta, c)$ for call c under a context assumption $\eta \in \mathbb{T}$ is called* consistent, *if it satisfies the following equation:*

Case: $\varphi = \mathbf{EX}\psi$

$$\alpha(\eta, c) = \psi \in \mu_{S_{\leftarrow c}}(l^{\mathrm{in}}(S_{\leftarrow c})) \tag{A.2}$$

Case: $\varphi \equiv \mathbf{EG}\psi \mid \mathbf{E}\psi\mathbf{U}\chi$

$$\alpha(\eta, c) = \omega_c(ca(\gamma, c, \eta), l^{\mathrm{in}}(S_{\leftarrow c})) \tag{A.3}$$

A summary γ is called consistent, *if $\alpha(\eta, c)$ is consistent for all $c \in C$ and all $\eta \in \mathbb{T}$.*
A summary π for $\mathcal{R}$ is called consistent, *if $\pi(S)$ is consistent for all $S \in \mathbb{S}$.*

For the case of $\varphi \equiv \mathbf{EX}\psi$, the assumption about a call reflects the labeling in the initial location of the called sub-structure with respect to ψ. For the other cases, consistency requires that the assumption about a call c matches the guarantee given in the initial location l^{in} of the called sub-structure. In this case, it is important to choose the correct context assumption. This is of course the context assumption *ca* for the call c.

The basic approach uses the equations in Definition A.2.5 to achieve consistency by replacing the equality with an assignment: Once all assumptions about all calls in all substructures and with all context assumptions are updated according to the equations, the resulting summary is certainly consistent.

A.2.7. The Summary Generation Algorithm

The algorithm for generating a fully labeled summary uses the definitions of maximal coherence and consistency to incrementally decide the guarantees and assumptions of the summary. Once a fixed point is reached, that is, all guarantees are maximally coherent and all assumptions are consistent, the remaining $\mathbf{m}$ values can be decided, thus yielding a fully decided summary.

Algorithm A.1 shows the function *subcheck* which generates a fully labeled summary. Its structure is basically equal to the *deduceLabels* function of the new approach. The function takes a recursive Kripke structure $\mathcal{R}$ and a formula φ as input and returns a fully labeled summary for φ. Note that $\mathcal{R}$ must already be labeled with respect to subformulae ψ of φ. This implies, that the checking for subformulae has to be accomplished in advance. Therefore, the checking must be performed bottom-up from the innermost subformulae to the formulae that use them.

The algorithm starts with the initialization of a summary π in which all guarantees and assumptions are initialized with $\mathbf{m}$. The functions *makeMaxCoherent* and *makeConsistent* use the definitions of maximal coherence and consistency, respectively, to update the values in π. The updates are executed alternatingly until the fixed point is found, which is the case once all assumptions are consistent after the guarantees have been made maximally

Algorithm A.1 *subcheck* : $\mathbb{R} \times \text{CTL} \rightarrow \Pi$

```
fun subcheck($\mathcal{R}, \varphi$) =
$\pi \leftarrow (\_ \mapsto \mathbf{m})$
$\pi \leftarrow makeMaxCoherent(\mathcal{R}, \varphi, \pi)$
while $\neg isConsistent(\mathcal{R}, \pi)$ do
    $\pi \leftarrow makeConsistent(\mathcal{R}, \pi)$
    $\pi \leftarrow makeMaxCoherent(\mathcal{R}, \varphi, \pi)$
end while
$\pi \leftarrow decideRemainingMaybes(\varphi, \pi)$
return $\pi$
```

coherent. The function *isConsistent* uses the definition of consistency to check when that is the case.

Once the loop terminates and the fixed point is found, there may still be remaining $\mathbf{m}$ guarantees and assumptions. These values are now resolved by the function *decideRemainingMaybes*. This step is equal to the one in the new algorithm: In case of **EU**, remaining $\mathbf{m}$ values are labeled $\mathbf{f}$. In case of **EG**, remaining $\mathbf{m}$ values are labeled $\mathbf{t}$. In case **EX**, no $\mathbf{m}$ labels can exist, because the first call of *makeMaxCoherent* is always able to label all locations in this case.

After the remaining values have been decided, the summary π, which is now fully labeled, is returned. It can be used to infer the result in the initial configuration or to check formulae which use φ as subformula. To do the checking of nested formulae, some more steps have to be performed, which are presented in the next section.

A.3. Checking Nested CTL Formulae

The previous section depicted the algorithm for creating a fully decided summary for a formula φ. A prerequisite for this algorithm was that the validity of subformulae of φ was present as atomic propositions in the RKS. This section presents the steps which have to be performed to check nested formulae. As already stated in the previous section, a bottom up approach is used which checks the innermost formulae (which are atomic propositions or $\top$) first and then uses the result to check outer formulae. The input rks $\mathcal{R}$ and summary generated by the *subcheck* function has to be transformed into a new rks $\mathcal{R}'$ which has its locations labeled with labels φ where φ is valid.

The resulting RKS has usually more sub-structures than the initial one, because the context assumptions are "woven" into the structure of the RKS: If a sub-structure is called with context assumption $\mathbf{t}$ and $\mathbf{f}$ by two different calls, then the sub-structure is split into two new sub-structures, one for context assumption $\mathbf{t}$ and one for $\mathbf{f}$, and the calls are pointed to the sub-structure which corresponds to their call context assumption. By weaving the context assumptions into the structure of the RKS itself after each checking of a formula, no formula context θ, as used by the new algorithm, is necessary. Instead, the context is implicitly represented in the structure of the resulting RKS.

There are two main challenges which have to be addressed during the construction of the resulting RKS: The first one is the step from a decided summary to an RKS which contains the results of that summary as labels and has the context assumptions represented in its structure. This step is called *split*, because it usually splits a sub-structure into two new ones with different context assumptions. All calls which previously called a split sub-structure now have to be "routed" to the correct resulting structure. The second challenge is the merging of the RKSs which are gathered from checking the subformulae of binary operators like $\mathbf{E}\psi\mathbf{U}\chi$ and $\psi \vee \chi$. In this step, the sub-structures of the two RKSs have to be merged into new resulting sub-structures. While the combining of the labels for ψ and χ is quite straight forward, the challenge lies again in the correct assignment of call targets. This step is called *merge*.

A.3.1. Split

During the split step, a new RKS is built from the input RKS and the generated summary π for a formula φ. The $\mathbf{t}$ guarantees in the summary are to be included as φ labels in the resulting RKS. The challenge in this step is that a summary always contains two guarantees for a location l, one for the $\mathbf{t}$ and one for the $\mathbf{f}$ context assumption[1]. The context assumption to be used depends on the validity of φ in the successor locations of a call which calls the sub-structure. This dependency can lead to a split of a sub-structure: If a sub-structure S is called by two calls and the formula holds in the successor locations of one of them but not in the successor locations of the other, then two sub-structures have to be constructed from S. One of them is labeled with the guarantees for the $\mathbf{f}$ context assumption and the other one for the $\mathbf{t}$ context assumption. The call target functions ν have to be altered to point to the respective newly created sub-structure.

The split operation is performed by doing an exploration of the call-graph which visits sub-structures under a given context assumption: It starts with the initial sub-structure S^{in} under the initial context assumption η^{in}. It then explores all calls recursively, using their respective call context assumption *ca*. Each visited sub-structure context assumption pair (S, η) is used to generate a new sub-structure. This sub-structure retains the structure of S but labels locations l according to the guarantees $\omega(\eta, l)$ of the summary using context assumption η. To avoid visiting a pair twice, all visited pairs are saved in a mapping which maps this pair to the sub-structure which was created for it.

Algorithm A.2 shows the pseudo-code for the *split* method. The method takes an RKS $\mathcal{R}$, a fully labeled summary π for a formula φ, and an initial context assumption η^{in}. It returns a new RKS $\mathcal{R}'$ which has the validity of φ encoded as φ labels.

The algorithm starts with calling the *exploreAndSplit* function which performs the call-graph exploration. This exploration is started with the initial sub-structure S^{in} called with the initial context assumption η^{in}. The exploration function uses an accumulator $\upsilon : \mathbb{S} \times \mathbb{T} \mapsto \mathbb{S}$ which stores the information about which configurations are already visited: It maps visited sub-structure context assumption pairs to the sub-structures which were generated for them. The accumulator starts empty, which means as a function which is undefined ($\perp$) for each sub-structure context assumption pair. This accumulator is also the return value of the exploration function.

[1] Actually, it contains three guarantees for the context assumptions $\mathbf{t}$, $\mathbf{f}$, and $\mathbf{m}$. However, the guarantees for $\mathbf{m}$ are only important during the summary generation where $\mathbf{m}$ assumptions still exist. After the generation, only the values for $\mathbf{t}$ and $\mathbf{f}$ are of further interest.

Algorithm A.2 *split* : $\mathbb{R} \times \Pi \times \mathbb{T} \to \mathbb{R}$

```
fun $split(\mathcal{R} \equiv (\mathbb{S}, S^{\text{in}}), \pi, \eta^{\text{in}}) =$
  $\upsilon \leftarrow exploreAndSplit((\_ \mapsto \bot), \pi, S^{\text{in}}, \eta^{\text{in}})$
  $\mathbb{S}' \leftarrow \emptyset$
  for all $(S, \eta, S') \in \upsilon$ do
    $\mathbb{S}' \leftarrow \mathbb{S}' \cup \{S'\}$
  end for
  $\mathcal{R}' \leftarrow (\mathbb{S}', \upsilon(S^{\text{in}}, \eta^{\text{in}}))$
  return $\mathcal{R}'$
fun $exploreAndSplit(\upsilon, \pi, S, \eta) =$
  if $\upsilon(S, \eta) = \bot$ then
    $S' \leftarrow copySub(S)$
    $\upsilon(S, \eta) \leftarrow S'$
    $(\alpha, \omega) \leftarrow \pi(S)$
    for all $l \in L(S)$ do    //Read labels from summary
      $\mu_{S'}(l) \leftarrow$ if $\omega(\eta, l) = \mathbf{t}$ then $\{\varphi\}$ else $\emptyset$
    end for
    for all $c \in C(S)$ do
      $\eta_c \leftarrow ca(\pi(S), c, \eta)$
      $\upsilon \leftarrow exploreAndSplit(\upsilon, \pi, S_{\leftarrow c}, \eta_c)$    //Recursive exploration
      $\nu_{S'}(c) \leftarrow \upsilon(S_{\leftarrow c}, \eta_c)$    //Set call target
    end for
  end if
  return $\upsilon$
```

The exploration function starts with checking whether the pair (S, η) is already visited (line 1). If it is, the function returns immediately. Otherwise, a new sub-structure is created from S and inserted into the accumulator υ (lines 2 and 3). The operation *copySub* creates a copy of S which has an empty call mapping and label function ($\nu(c) = \bot$ for all c, $\mu(l) = \emptyset$ for all l). The rest of the exploration function fills ν and μ with the respective values. First, the labels are assigned to μ by labeling a location l with φ if the guarantee $\omega(\eta, l)$ is $\mathbf{t}$ (lines 4 to 7). Afterwards, the recursive exploration of calls in $C(S)$ is performed (line 8 to 12). For each call, the context assumption η_c is calculated using the *ca* function (line 9). Then, the called sub-structure $S_{\leftarrow c}$ is visited using η_c as context assumption (line 10). After the target of call c is visited, the sub-structure which was generated for that target is inserted as call target for this call (line 11).

After the exploration is finished, the *split* function iterates over the accumulator and inserts the generated sub-structures S' into a new set $\mathbb{S}'$. A new RKS $\mathcal{R}'$ is built from $\mathbb{S}'$ in which the initial sub-structure is the sub-structure generated for the initial sub-structure of the input RKS under the initial context assumption ($\upsilon(S^{\text{in}}, \eta^{\text{in}})$).

As visible from the algorithm, the "weaving" of the call context into the structure of the RKS is performed in line 11 of function *exploreAndSplit*: The target of a call is redirected to the sub-structure which was built for the corresponding context assumption. Therefore, two calls which previously pointed to the same sub-structure can point to different ones after the split, if they called the sub-structure under different context assumptions. This is the way the basic approach encodes the call context implicitly in the structure of the resulting RKS.

A.3.2. Merge

During the checking of nested formulae, there are cases where two recursive Kripke structures have to be combined. Concretely, this is the case whenever a formula with two subformulae is checked, that is, the cases of $\varphi \equiv \psi \vee \chi$ and $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$. The checking of the subformulae ψ and χ each yields a recursive Kripke structure. However, the summary generation algorithm expects one recursive Kripke structure which has both subformulae as atomic labels. Since the basic algorithm encodes the context in the structure itself, it is necessary to merge the structure of the RKSs to combine the contexts. The *merge* operation accomplishes this transformation: It merges the labels of two RKSs into one RKS with both labels.

Just like the *split* operation, a merge is conducted by performing a call-graph exploration. Since there are two call graphs now (the two input RKSs), these have to be explored synchronously to decide, which sub-structures have to be merged. The merge operation is depicted in Algorithm A.3.

The structure of the function is very similar to the one of the split operation. The first step is to perform the exploration (*exploreAndMerge*) starting from the respective initial sub-structure of the two input RKSs. In this case, the accumulator $\upsilon : \mathbb{S} \times \mathbb{S} \to \mathbb{S}'$ is a mapping from two sub-structures (one of each input RKS) to the resulting sub-structure which was generated for this pair of sub-structures.

The *exploreAndMerge* starts by creating a new sub-structure S' from any of the two sub-structures. Again, the call mapping and label function are empty for the newly created sub-structure (line 2). This sub-structure is inserted into the accumulator (line 3). Afterwards,

Algorithm A.3 *merge* : $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$

```
fun merge($\mathcal{R}_1 \equiv (\mathbb{S}_1, S_1^{\text{in}}), \mathcal{R}_2 \equiv (\mathbb{S}_2, S_2^{\text{in}})$) =
 1: $\upsilon \leftarrow$ exploreAndMerge($(\_ \mapsto \bot), S_1^{\text{in}}, S_2^{\text{in}}$)
 2: $\mathbb{S}' \leftarrow \emptyset$
 3: for all $(S_1, S_2, S') \in \upsilon$ do
 4:     $\mathbb{S}' \leftarrow \mathbb{S}' \cup \{S'\}$
 5: end for
 6: $\mathcal{R}' \leftarrow (\mathbb{S}', \upsilon(S_1^{\text{in}}, S_2^{\text{in}}))$
 7: return $\mathcal{R}'$
fun exploreAndMerge($\upsilon, S_1, S_2$) =
 1: if $\upsilon(S_1, S_2) = \bot$ then
 2:     $S' \leftarrow$ copySub($S_1$)
 3:     $\upsilon(S_1, S_2) \leftarrow S'$
 4:     for all $l \in L(S)$ do    //Merge labels
 5:         $\mu_{S'}(l) \leftarrow \mu_{S_1}(l) \cup \mu_{S_2}(l)$
 6:     end for
 7:     for all $c \in C(S)$ do
 8:         $\upsilon \leftarrow$ exploreAndMerge($\upsilon, S^1_{\leftarrow c}, S^2_{\leftarrow c}$)    //Recursive exploration ($S^x_{\leftarrow c} = \nu_{S_x}(c)$)
 9:         $\nu_{S'}(c) \leftarrow \upsilon(S^1_{\leftarrow c}, S^2_{\leftarrow c})$    //Set call target
10:     end for
11: end if
12: return $\upsilon$
```

the labels of the new sub-structure are generated by merging the labels of the input sub-structures (line 4 to 6). Finally, recursive exploration of calls is performed and the call targets are set to the newly created sub-structures (lines 7 to 11). After the exploration function returns the accumulator, the *merge* function builds a new RKS $\mathcal{R}'$ from the created sub-structures, using the sub-structure which was created for the two initial sub-structures of the input RKSs as initial sub-structure.

A merge yields the combination of two RKSs for subformulae ψ and χ which can be used to check, for example, $\psi \vee \chi$. Using this operation and the split operation from the previous section, the overall approach for checking nested formulae can be defined.

A.3.3. Labeling Nested Formulae

The last two sections introduced the operations *split* and *merge* which generate an RKS from a summary and merge two RKSs, respectively. Of course, a third fundamental operation for the checking nested formulae is the summary generation function *subcheck* which was elaborated in Section A.2.7. As already mentioned, the algorithm for checking nested formulae is performed by structural recursion. The function for conducting the check is called *label* and is depicted by Algorithm A.4.

The function takes a recursive Kripke structure $\mathcal{R}$ and a CTL formula φ. Of course, it is a requirement that φ and $\mathcal{R}$ must be defined over the same set of atomic proposition. The result is a pair of an RKS which contains labels representing the validity of φ and an initial

Algorithm A.4 *label* : $\mathbb{R} \times \mathrm{CTL} \rightarrow (\mathbb{R} \times \mathbb{T})$

The *label* function is defined recursively over the structure of the formula φ which is to be checked.

```
fun label($\mathcal{R}, \varphi$) =
  $\varphi \equiv \top \rightarrow$
      $\mathcal{R}^{\varphi} \leftarrow$ relabel all $S \in \mathbb{S}(\mathcal{R}), l \in L(S)$ set $\mu_S(l) = \{\top\}$
      return $(\mathcal{R}^{\varphi}, \mathbf{t})$
  $\varphi \equiv p \in AP \rightarrow$
      return $(\mathcal{R}, \mathbf{f})$
  $\varphi \equiv \neg\psi \rightarrow$
      $(\mathcal{R}^{\psi}, \eta^{\psi}) \leftarrow label(\mathcal{R}, \psi)$
      $\mathcal{R}^{\varphi} \leftarrow$ relabel all $S \in \mathbb{S}(\mathcal{R}^{\psi}), l \in L(S)$
          set $\mu_S(l) = \{\neg\psi\}$ where $\psi \notin \mu_S(l)$
      return $(\mathcal{R}^{\varphi}, \neg\eta^{\psi})$
  $\varphi \equiv \psi \vee \chi \rightarrow$
      $(\mathcal{R}^{\psi}, \eta^{\psi}) \leftarrow label(\mathcal{R}, \psi)$
      $(\mathcal{R}^{\chi}, \eta^{\chi}) \leftarrow label(\mathcal{R}, \chi)$
      $\mathcal{R}' \leftarrow merge(\mathcal{R}^{\psi}, \mathcal{R}^{\chi})$
      $\mathcal{R}^{\varphi} \leftarrow$ relabel all $S \in \mathbb{S}(\mathcal{R}'), l \in L(S)$
          set $\mu_S(l) = \{\psi \vee \chi\}$ where $\psi \in \mu_S(l) \vee \chi \in \mu_S(l)$
      return $(\mathcal{R}^{\varphi}, \eta^{\psi} \vee \eta^{\chi})$
  $\varphi \equiv \mathbf{EX}\psi \mid \mathbf{EG}\psi \rightarrow$
      $(\mathcal{R}^{\psi}, \eta^{\psi}) \leftarrow label(\mathcal{R}, \psi)$
      $\pi^{\varphi} \leftarrow subcheck(\mathcal{R}^{\psi}, \varphi)$
      $\mathcal{R}^{\varphi} \leftarrow split(\mathcal{R}^{\psi}, \pi^{\varphi}, \eta^{\psi})$
      return $(\mathcal{R}^{\varphi}, \eta^{\psi})$
  $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi \rightarrow$
      $(\mathcal{R}^{\psi}, \eta^{\psi}) \leftarrow label(\mathcal{R}, \psi)$
      $(\mathcal{R}^{\chi}, \eta^{\chi}) \leftarrow label(\mathcal{R}, \chi)$
      $\mathcal{R}' \leftarrow merge(\mathcal{R}^{\psi}, \mathcal{R}^{\chi})$
      $\pi^{\varphi} \leftarrow subcheck(\mathcal{R}', \varphi)$
      $\mathcal{R}^{\varphi} \leftarrow split(\mathcal{R}', \pi^{\varphi}, \eta^{\chi})$
      return $(\mathcal{R}^{\varphi}, \eta^{\chi})$
```

context assumption η. This context assumption represents the context under which the initial sub-structure is called in the initial configuration. The context assumption returned by the algorithm is equal to the one of the new approach as defined by the *ica* function (cf. page 37).

The base cases for the recursion are the ones where φ is either $\top$ or where it is an atomic proposition p. In the case of an atomic proposition, the Kripke structure is simply returned with an $\mathbf{f}$ initial context assumption. In case of $\top$, $\mathcal{R}$ is *relabeled*. The result of such a relabel operation is an RKS which shares all components with the source RKS, except of the labeling functions μ_S, which are replaced. In this case, a $\top$ label is inserted at each location. The context assumption for $\top$ is $\mathbf{t}$, of course.

The first recursive case is the one where $\varphi \equiv \neg\psi$. Each recursive step begins with the recursive checking of subformulae. In this case, the subformula ψ is checked, which yields an RKS $\mathcal{R}^{\psi}$ and a initial context assumption η^{ψ}. The resulting structure is again obtained by performing a relabeling, which "inverses" the labels: In each location l where the source structure had the label ψ, the new structure has no label. In all other locations, the new structure receives the label $\neg\psi$. The "negated" RKS is returned, together with the initial context assumption η^{ψ} of the subformula, which is negated as well.

The second recursive case is $\varphi \equiv \psi \vee \chi$. Again, the first step is the recursive checking of the subformulae. Since the operator $\vee$ is a binary one, a merge has to be performed to merge the labels of the RKSs for ψ and χ into one RKS. After the merge, a relabeling is conducted which performs a logical disjunction of the labels: Each location which has at least one of the labels ψ or χ receives the label $\psi \vee \chi$. All other locations receive no label. The relabeled structure is returned together with the initial context assumptions of the subformulae, which are combined by a logical disjunction, as well.

In all of the mentioned cases, no summaries are necessary, because they merely represent propositional logic operations. The remaining cases are path quantifiers which implies that they require the generation of a summary and the split operation.

The first path quantifier case is the one where φ is either $\mathbf{EX}\psi$ or $\mathbf{EG}\psi$. After the recursive call for the subformula ψ, φ is checked by generating a fully labeled summary. The split operation transforms the results from the summary to a new RKS which is labeled accordingly. This RKS is returned together with the initial context assumption of the subformula, which is not altered.

The final case is $\varphi \equiv \mathbf{E}\psi\mathbf{U}\chi$. After the recursive check of the subformulae, the resulting RKSs are combined with the merge operation, yielding an RKS which can be used as input for the *subcheck* algorithm. The rest of this case is equivalent to the one before: The summary is generated and split is used to create a properly labeled RKS from it. This RKS is returned, together with the initial context assumption of the χ subformula.

Using the *label* function, an arbitrarily nested formula can be labeled for an RKS. For model checking an RKS $\mathcal{R}$ with respect to a formula φ, the *label*$(\mathcal{R}, \varphi)$ function is executed and the resulting rks $\mathcal{R}' = (\mathbb{S}, S^{\text{in}})$ is used to deduce the validity of φ in the initial configuration of the RKS:

$$\mathcal{R} \models \varphi \Leftrightarrow \varphi \in \mu_{S^{\text{in}}}(l^{\text{in}}(S^{\text{in}}))$$

A.4. Correctness

Because the basic algorithm is not in the direct focus of this thesis, no proof for its correctness is given. However, a draft for how a proof could look like is suggested here.

The labeling of **EU** and **EG** with the fixed point algorithm is done in a similar way as already proven for the new approach. Therefore, a proof for this step will also look very similar to the one conducted. The only difference is that more than one successor location of a call may exist in the basic algorithm, so any proof containing statements about properties of the successor location must instead be conducted for any of the successor locations. This should not pose a problem, since both operators are existentially quantified. Therefore, it suffices to have one path (i.e., one of the successor locations) satisfying a property. This is exactly what the *ca* function does by computing the logical disjunction of the formula validity values in the successor locations.

The labeling for **EX** is done differently than in the new approach. However, it is trivial to show its correctness. The same is true for the non-temporal operators.

The only aspect which really needs attention is the split and the merge step. These steps to create new RKSs are not performed in the new approach. Instead, the new approach works with a full call context θ_φ which represents assumptions about the validity of φ and each of its subformulae in the successor location of a call. A proof for the correctness of split and merge could be conducted by showing that these operations insert the call context into the structure of the RKS correctly. It is quite certain that this is the case, because especially split does "link" all call targets to the sub-structure with the correct context assumption and therefore represent the context assumption for the topmost formula by choosing the respective call target. As this is done for each formula, the topmost context assumption is always inserted into the structure of the RKS. Thus, a proof could be conducted by structural induction over the formula to show that the full call context θ_φ is represented in the link structure of the RKS. If this is shown, then the operations split and merge are basically equal to the steps conducted in new approach. Therefore, the correctness of the new approach could be used to argue that also the basic approach is correct.

Bibliography

[1] R. Alur. Model checking: From tools to theory. *25 Years of Model Checking*, pages 89–106, 2008.

[2] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):786–818, 2005.

[3] R. Alur, S. Chaudhuri, K. Etessami, and P. Madhusudan. On-the-fly reachability and cycle detection for recursive state machines. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 61–76, 2005.

[4] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–481, 2004.

[5] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Computer Aided Verification*, pages 207–220. Springer, 2001.

[6] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. *Automata, Languages and Programming*, pages 703–703, 1999.

[7] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211. ACM, 2004.

[8] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Language Theory*, pages 1–13. Springer, 2006.

[9] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 175–188. ACM, 1998.

[10] B. Aminof, A. Murano, and M. Vardi. Pushdown module checking with imperfect information. *CONCUR 2007–Concurrency Theory*, pages 460–475, 2007.

[11] C. Baier and J.P. Katoen. *Principles of model checking*, volume 950. MIT press, 2008.

[12] T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic predicate abstraction of c programs. In *ACM SIGPLAN Notices*, volume 36, pages 203–213. ACM, 2001.

[13] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. *SPIN Model Checking and Software Verification*, pages 113–130, 2000.

[14] T. Ball and S. Rajamani. The slam toolkit. In *Computer aided verification*, pages 260–264. Springer, 2001.

[15] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122. Springer-Verlag New York, Inc., 2001.

[16] T. Ball and S.K. Rajamani. The s lam project: debugging system software via static analysis. *ACM SIGPLAN Notices*, 37(1):1–3, 2002.

[17] G. Basler, D. Kroening, and G. Weissenbacher. Sat-based summarization for boolean programs. In *Proceedings of the 14th international SPIN conference on Model checking software*, pages 131–148. Springer-Verlag, 2007.

[18] G. Basler, D. Kroening, and G. Weissenbacher. A complete bounded model checking algorithm for pushdown systems. *Hardware and Software: Verification and Testing*, pages 202–217, 2008.

[19] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. *Automata, Languages and Programming*, pages 652–666, 2001.

[20] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. The software model checker b last. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5):505–525, 2007.

[21] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. *CONCUR'97: Concurrency Theory*, pages 135–150, 1997.

[22] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *ACM SIGPLAN Notices*, volume 38, pages 62–73. ACM, 2003.

[23] L. Bozzelli. Complexity results on branching-time pushdown model checking. *Theoretical computer science*, 379(1-2):286–297, 2007.

[24] L. Bozzelli, A. Murano, and A. Peron. Pushdown module checking. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 504–518. Springer, 2005.

[25] T. Brázdil, A. Kučera, and O. Stražovskỳ. On the decidability of temporal properties of probabilistic pushdown automata. *STACS 2005*, pages 145–157, 2005.

[26] O. Burkart and Y.M. Quemener. *Model-checking of infinite graphs defined by graph grammars*. Citeseer, 1996.

[27] O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR'92*, pages 123–137. Springer, 1992.

[28] O. Burkart and B. Steffen. Composition, decomposition and model checking of pushdown processes. *Nordic Journal of Computing*, 2(2):89–125, 1995.

[29] O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Automata, Languages and Programming*, pages 419–429, 1997.

[30] T. Cachat. Two-way tree automata solving pushdown games. *Automata logics, and infinite games*, pages 413–419, 2002.

[31] T. Cachat. Higher order pushdown automata, the caucal hierarchy of graphs and parity games. *Automata, Languages and Programming*, pages 193–193, 2003.

[32] T. Cachat. Uniform solution of parity games on prefix-recognizable graphs. *Electronic Notes in Theoretical Computer Science*, 68(6):71–84, 2003.

[33] A. Carayol, M. Hague, A. Meyer, C.H.L. Ong, and O. Serre. Winning regions of higher-order pushdown games. In *Logic in Computer Science, 2008. LICS'08. 23rd Annual IEEE Symposium on*, pages 193–204. IEEE, 2008.

[34] A. Carayol and S. Wöhrle. The caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science*, pages 112–123, 2003.

[35] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *ACM SIGPLAN Notices*, volume 43, pages 159–169. ACM, 2008.

[36] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244. ACM, 2002.

[37] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 241–268. Springer, 2002.

[38] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *Computer Aided Verification*, pages 682–682. Springer, 1999.

[39] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, pages 232–247. Springer, 2000.

[40] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural dataflow analysis. In *Foundations of Software Science and Computation Structures*, pages 642–642. Springer, 1999.

[41] J. Esparza, A. Kucera, and R. Mayr. Model checking probabilistic pushdown automata. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 12–21. IEEE, 2004.

[42] J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *Computer Aided Verification*, pages 324–336. Springer, 2001.

[43] K. Etessami and M. Yannakakis. Algorithmic verification of recursive probabilistic state machines. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 253–270, 2005.

[44] A. Fehnker and C. Dubslaff. Inter-procedural analysis of computer programs, February 19 2010. US Patent App. 12/709,053.

[45] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Goanna—a static model checker. *Formal Methods: Applications and Technology*, pages 297–300, 2007.

[46] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Model checking software at compile time. In *Theoretical Aspects of Software Engineering, 2007. TASE'07. First Joint IEEE/IFIP Symposium on*, pages 45–56. IEEE, 2007.

[47] A. Ferrante, A. Murano, and M. Parente. Enriched μ-calculus pushdown module checking. In *Proceedings of the 14th international conference on Logic for programming, artificial intelligence and reasoning*, pages 438–453. Springer-Verlag, 2007.

[48] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9:27–37, 1997.

[49] S. Gnesi and F. Mazzanti. On the fly model checking of communicating uml state machines. In *Second ACIS International Conference on Software Engineering Research, Management and Applications (SERA2004)*, pages 331–338, 2004.

[50] M. Hague. *Saturation methods for global model-checking pushdown systems*. PhD thesis, PhD. Thesis, University of Oxford, 2009.

[51] M. Hague and C. Ong. Symbolic backwards-reachability analysis for higher-order pushdown systems. *Foundations of Software Science and Computational Structures*, pages 213–227, 2007.

[52] G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[53] K. Hristova and Y. Liu. Improved algorithm complexities for linear temporal logic model checking of pushdown systems. In *Verification, Model Checking, and Abstract Interpretation*, pages 190–206. Springer, 2006.

[54] R. Huuck, A. Fehnker, S. Seefried, and J. Brauer. Goanna: Syntactic software model checking. *Automated Technology for Verification and Analysis*, pages 216–221, 2008.

[55] S. Kripke. Semantical considerations on modal logic. *Acta philosophica fennica*, 16(1963):83–94, 1963.

[56] A. Kucera, J. Esparza, and R. Mayr. Model checking probabilistic pushdown automata. *Logical Methods in Computer Science*, 2(1), 2006.

[57] O. Kupferman, N. Piterman, and M. Vardi. Model checking linear properties of prefix-recognizable systems. In *Computer Aided Verification*, pages 281–302. Springer, 2002.

[58] O. Kupferman, N. Piterman, and M. Vardi. An automata-theoretic approach to infinite-state systems. *Time for verification*, pages 202–259, 2010.

[59] O. Kupferman and M. Vardi. Module checking. In *Computer Aided Verification*, pages 75–86. Springer, 1996.

[60] O. Kupferman and M. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Computer Aided Verification*, pages 36–52. Springer, 2000.

[61] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM (JACM)*, 47(2):312–360, 2000.

[62] S. La Torre, M. Napoli, M. Parente, and G. Parlato. Verification of scope-dependent hierarchical state machines. *Information and Computation*, 206(9-10):1161–1177, 2008.

[63] J. Obdrzálek. Model checking java using pushdown systems. In *Workshop on Formal Techniques for Java-like Programs*. Citeseer, 2002.

[64] N. Piterman and M. Vardi. Global model-checking of infinite-state systems. In *Computer Aided Verification*, pages 311–314. Springer, 2004.

[65] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[66] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[67] S. Schwoon. *Model-checking pushdown systems*. PhD thesis, Technische Universität München, Universitätsbibliothek, 2002.

[68] F. Song and T. Touili. Efficient ctl model-checking for pushdown systems. *CONCUR 2011–Concurrency Theory*, pages 434–449, 2011.

[69] D. Suwimonteerabuth, F. Berger, S. Schwoon, and J. Esparza. jmoped: A test environment for java programs. In *Proceedings of the 19th international conference on Computer aided verification*, pages 164–167. Springer-Verlag, 2007.

[70] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jmoped: A java bytecode checker based on moped. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 541–545, 2005.

[71] M. Vardi. Reasoning about the past with two-way automata. *Automata, Languages and Programming*, pages 628–641, 1998.

[72] I. Walukiewicz. Pushdown processes: Games and model checking. In *Computer Aided Verification*, pages 62–74. Springer, 1996.

[73] I. Walukiewicz. Model checking ctl properties of pushdown systems. *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, pages 127–138, 2000.

List of Figures

List of Algorithms