Producing Explanations for Rich Logics

Simon Busard and Charles Pecheur

Université catholique de Louvain, Louvain-la-Neuve, Belgium,
{simon.busard,charles.pecheur}@uclouvain.be

Abstract. One of the claimed advantages of model checking is its capability to provide a counter-example explaining why a property is violated by a given system. Nevertheless, branching logics such as Computation Tree Logic and its extensions have complex branching counter-examples, and standard model checkers such as NuSMV do not produce complete counter-examples—that is, counter-examples providing all information needed to understand the verification outcome—and are limited to single executions. Many branching logics can be translated into the µ-calculus. To solve this problem of producing complete and complex counter-examples for branching logics, we propose a µ-calculus-based framework with rich explanations. It integrates a µ-calculus model checker that produces complete explanations, and several functionalities to translate them back to the original logic. In addition to the framework itself, we describe its implementation in Python and illustrate its applicability with Alternating Temporal Logic.

1 Introduction

Model checking is a verification technique that performs an exhaustive search among the behaviors of a system to determine if it satisfies a given property, usually expressed in a logic [10,2]. *Branching logics*, such as CTL, express properties about the branching structure of the system [12]. Many extensions of CTL have been proposed to take into account other aspects of the verified systems, such as knowledge—with CTLK [28]—, or strategic abilities—with ATL [1]. Such logics can be translated into the *propositional μ-calculus*, a logic based on fixpoint and modal operators [22].

Producing an explanation of the verification outcome is one of the claimed advantages of model checking. But, in the case of branching logics, the explanations can be very rich as, in general, branching logics need branching counter-examples [3]. They have to show different branches of the execution tree of the system to fully explain the truth value of the property. However, current state-of-the-art tools such as NuSMV only produce single executions of the model when explaining why a property is violated [9].

The goal of this paper is to propose techniques and tools to generate, visualize and manipulate explanations for µ-calculus-based logics such as CTL, CTLK and ATL. Let us suppose that someone—the *designer*—uses some logic—the *top-level logic*—to express and verify facts about some system, and wants to develop a model checker for it. She can either develop the tool from scratch, or she can translate the models and formulas into another logic—the *base logic*—and use existing tools to solve the model-checking problem.

Many logics can be translated into the µ-calculus, making it a good candidate for a base logic. Nevertheless, when translating her model-checking problem into µ-calculus, the designer has no help to facilitate this translation, in particular, the counter-examples returned by the model checker (if any) are expressed in terms of µ-calculus primitives instead of top-level logic ones. To overcome this limitation and to help designers to quickly develop a model checker with rich counter-examples, this paper proposes a µ-calculus-based framework with rich explanations. The framework provides a µ-calculus model checker that generates rich explanations and functionalities to define how top-level logic formulas are translated into µ-calculus, to control how the µ-calculus explanations are generated, and to translate µ-calculus explanations into top-level logic ones. These functionalities are the following:

1. *Formula aliases* link the formulas stored in the obligations to the top-level logic formulas they represent.
2. The *relational graph algebra* of Dong et al. [15] is provided to transform explanations into the part of the original model they represent.
3. *Obligation and edge attributors* add information to individual nodes and edges of the explanation graph.
4. *Local translation* focuses on the small part that explains a given alias without having to deal with the whole graph at once.
5. *Choosers* can be used to perform interactive or guided generation of explanations. They also introduce the notion of *partial explanations*.
6. *Formula markers* are tags on formulas. Points of interest and points of decision are provided, but other markers can be defined by the designer.

All these functionalities work together to help the designer to produce useful explanations. Figure 1 illustrates the structure of the framework.

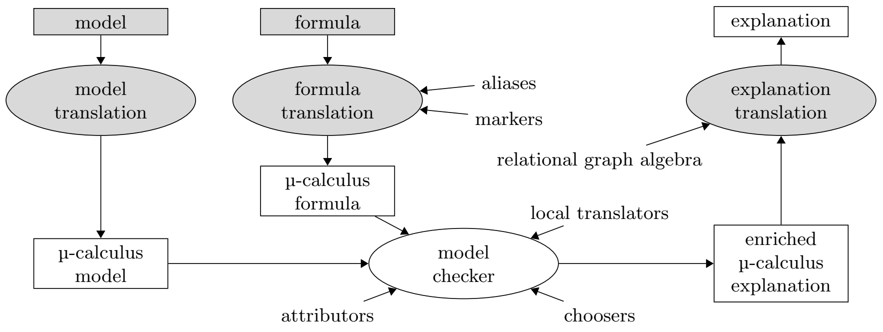


Fig. 1. The structure of the framework. In gray, the parts that the designer has to define; in white, the elements provided by the framework.

The designer first translates the original model and formula into µ-calculus. She can decorate the translated formulas with aliases and markers, and she can also attach attributors, local translators and choosers. The aliases and markers will be present in the obligations in the generated enriched µ-calculus explanation to help the designer with their translation. The attributors and local translators are used by the model checker to add extra information to the generated explanations. The choosers allow the model checker to make the right choices. Finally, the designer translates the enriched explanation back into the top-level logic language thanks to the relational graph algebra.

The features are generic and complement each other: (1) the relational algebra, attributors and local translators manipulate the explanation at different scales; (2) points of decision and choosers work together to produce smaller partial strategies and to select the explanations of interest; (3) points of interest and aliases add information to important formulas.

The remainder of this paper is structured as follows: Sect. 2 presents the propositional µ-calculus. Section 3 describes the framework for µ-calculus-based logics explanations, and Sect. 4 its implementation in Python. Section 5 applies the framework to the case of ATL model checking. Section 6 briefly compares the framework with related work, and Sect. 7 draws conclusions.

2 The Propositional µ-Calculus

The µ-calculus is a logic based on fixpoints [22]. Its formulas follow the grammar

$$\phi ::= \mathit{true} \mid p \mid v \mid \neg\phi \mid \phi \vee \phi \mid \Diamond_i\ \phi \mid \mu v.\ \phi$$

where $p \in AP$ are atomic propositions and $v \in \mathit{Var}$ are variables. For instance, $\Diamond_i\ \phi$ means that *there exists a successor through the transition relation i that satisfies ϕ, that is, a state satisfying ϕ can be reached in one step through the transition relation i*.

We write $\mathcal{L}_\mu$ for the set of µ-calculus formulas. Other operators can be defined in terms of the ones above, such as $\Box_i\ \phi \equiv \neg \Diamond_i\ \neg\phi$ and $\nu v.\ \phi \equiv \neg\mu v.\ \neg\phi(\neg v)$.

A variable v is *bound* in ϕ if it is enclosed in a sub-formula $\mu v.\ \psi$ or $\nu v.\ \psi$; otherwise, it is *free*. We sometimes note $\mu v.\ \psi(v)$, $\nu v.\ \psi(v)$, and $\psi(v)$ to stress the fact that ψ contains free occurrences of variable v. We write $\psi[\chi/v]$—or equivalently $\psi(\chi)$ when v is clear from the context—for the µ-calculus formula ψ where every free occurrence of v is replaced by χ. We write $\psi^k(\chi)$ for k nestings of ψ around χ, that is, $\psi^0(\chi) = \chi$ and $\psi^{k+1}(\chi) = \psi(\psi^k(\chi))$.

Any formula $\mu v.\ \psi$ or $\nu v.\ \psi$, must be *syntactically monotone*, that is, all occurrences of v in ψ must fall under an even number of negations. A formula is in *positive normal form* if negations are only applied to atomic propositions and variables. Any syntactically monotone formula can be transformed into an equivalent syntactically monotone formula in positive normal form.

µ-calculus models are Kripke structures $S = \langle Q, \{R_i \mid i \in \Sigma\}, V\rangle$ where (1) Q is a finite set of states; (2) $R_i \subseteq Q \times Q$ are $|\Sigma|$ transition relations; (3) $V : Q \to 2^{AP}$ labels the states with atomic propositions. We write $q \to_i q'$ for $\langle q, q'\rangle \in R_i$.

µ-calculus formulas are interpreted as sets of states under a given environment. An environment is a function $e : \mathit{Var} \to 2^Q$ associating sets of states to variables. The set of environments is noted $\mathcal{E}$. We write $e[Q'/v]$, for $Q' \subseteq Q$ and $v \in \mathit{Var}$, for the function e' such that $e'(v) = Q'$ and e' agrees with e for all other variables. The semantics of formulas is given by the function $[\![\phi]\!]^S e$. It takes a formula ϕ and an environment e defined at least for the free variables of ϕ, and returns the corresponding set of states. This function is defined as:

$$
\begin{aligned}
&[\![\mathit{true}]\!]^S e = Q, && [\![\neg\phi]\!]^S e = Q \backslash [\![\phi]\!]^S e,\\
&[\![v]\!]^S e = e(v), && [\![\phi \vee \psi]\!]^S e = [\![\phi]\!]^S e \cup [\![\psi]\!]^S e,\\
&[\![p]\!]^S e = \{q \in Q \mid p \in V(q)\}, && [\![\mu v.\ \phi]\!]^S e = \bigcap\{Q' \subseteq Q \mid [\![\phi]\!]^S e[Q'/v] \subseteq Q'\}.\\
&[\![\Diamond_i\ \phi]\!]^S e = \{q \in Q \mid \exists q' \in Q \text{ s.t. } q \rightarrow_i q' \wedge q' \in [\![\phi]\!]^S e\},
\end{aligned}
$$

3 A µ-Calculus-based Framework for Rich Explanations

This section presents the µ-calculus-based framework we propose. To illustrate the concepts, we will use the case of ATL model checking, presented in Sect. 3.1. Section 3.2 describes µ-calculus explanations, and Sect. 3.3 presents the functionalities to translate these explanations back to the original logic.

3.1 Translation of ATL models and formulas to µ-calculus

ATL formulas are built with atomic propositions and Boolean connectives, as well as coalition modalities $\langle\!\langle\rangle\!\rangle$ and $[\![]\!]$ reasoning about the strategies of groups of agents to enforce temporal objectives specified with the standard **X**, **F**, **G** and **U** temporal operators [1]. For instance, the formula $\langle\!\langle \Gamma \rangle\!\rangle\mathbf{F}\ p$ expresses the fact that agents Γ have a strategy to reach, within a finite number of steps, some goal p, and $[\![\Gamma]\!]\mathbf{G}\ q$ that they have no strategy to maintain some other goal q forever.

ATL formulas are interpreted over the states of *concurrent game structures* (CGS) $S = \langle Ag, Q, Q_0, Act, e, \delta, V\rangle$ defining the states (Q) and agents (Ag) of the system, what they can do ($e : Ag \to (Q \to (2^{Act} \backslash \varnothing))$), and how the system evolves according to their choices ($\delta : Q \times Act^{Ag} \to Q$).

Given a CGS S, a state q of S, and an ATL formula ϕ, we can translate S into a Kripke structure S', q into a state q' of S', and ϕ into a µ-calculus formula ϕ' such that q satisfies ϕ if and only if q' satisfies ϕ'. To avoid technical details, this section only presents the intuition of the translation and focuses on a small subset of ATL operators. The full translation can be found in [4].

The idea of the translation from a CGS $S = \langle Ag, Q, Q_0, Act, e, \delta, V\rangle$ to a structure $S' = \langle Q', \{R'_i \mid i \in \Sigma\}, V'\rangle$ is to derive, from each state $q \in Q$, each group of agents $\Gamma \subseteq Ag$, and each joint action a_Γ of Γ, a new state q_{a_Γ} representing the fact that Γ chose to play a_Γ in q. For each group $\Gamma \subseteq Ag$, two transition relations are derived from δ: $R_{\Gamma choose}$ links any state $q \in Q$ to the derived states q_{a_Γ} for all possible actions a_Γ of Γ; $R_{\Gamma follow}$ links any derived state q_{a_Γ} to the successors of

q restricted to the ones reached if Γ choose a_Γ. Intuitively, the derived structure S' encodes in two steps ($q \to q_{a_\Gamma} \to q'$) the one-step transitions of S ($q \xrightarrow{a} q'$). The set Σ of relations names is $\Sigma = \{\Gamma choose \mid \Gamma \subseteq Ag\} \cup \{\Gamma follow \mid \Gamma \subseteq Ag\}$, that is, two transition relations for each group of agents.

Figure 2 presents the CGS of a simple one-bit transmission problem in which a *sender* tries to send a value through an unreliable *link*. The sender can *send* the value or *wait*, and the transmitter can *transmit* the message (if any), or *block* the transmission. In this context, we ask whether *the transmitter has a strategy to never transmit the value*, that is, if q_0 satisfies $\langle\langle transmitter\rangle\rangle \mathbf{G}\ \neg sent$.

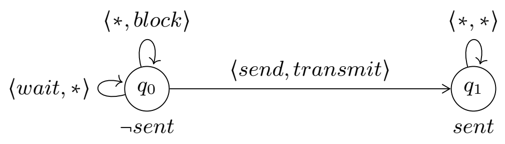


Fig. 2. The CGS of the bit transmission problem. The action pairs are the actions of the sender and the transmitter, respectively. $*$ means *any action of the agent.*

The CGS of this bit transmission problem can be translated into a μ-calculus Kripke structure. Figure 3 presents a part of the translation, focusing on the states derived from q_0; the part about q_1 is not shown. For instance, in q_0, the sender can choose the action *send* to transition to $q_{0_{send}}$. The transmitter's following action can either be *block*, which transitions back to q_0, or *transmit*, which transitions to q_1.

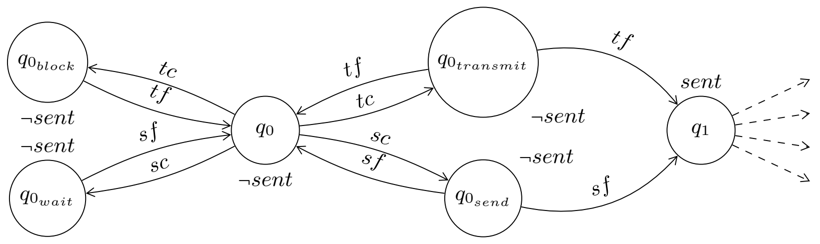


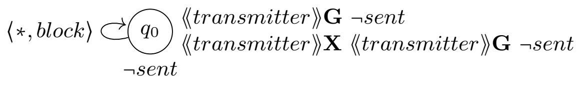
Fig. 3. A part of the translation of the bit transmission CGS. sc and sf mean *sender chooses* and *sender follows*, tc and tf mean *transmitter chooses* and *transmitter follows*. Transition relations for the two other groups of agents (no agent, and both agents) are not shown.

ATL formulas can also be translated into μ-calculus formulas. The formula $\langle\langle transmitter\rangle\rangle \mathbf{G}\ \neg sent$ is translated as

$$\phi_{ns} = \nu v.\ \neg sent \wedge \Diamond_{trans\ chooses} (\Diamond_{trans\ follows}\ true \wedge \Box_{trans\ follows}\ v). \quad (1)$$

The main idea behind this translation is that a state satisfies the second term $\Diamond_{trans\ chooses}\ (\Diamond_{trans\ follows}\ true \wedge \Box_{trans\ follows}\ v)$ if there exists an action for *transmitter* that is enabled and such that all choices of the other agents lead to v, that is, if the transmitter can enforce to reach v in one step. Then, a state satisfies ϕ_{ns} if the transmitter can enforce to stay in states satisfying $\neg sent$ forever, that is, if the transmitter has a strategy to enforce $\mathbf{G}\ \neg sent$.

To explain why an ATL formula ϕ is satisfied by a state q of some CGS S, we want to extract the part of the model starting at q that is responsible for the satisfaction. Furthermore, as such part can be complex and difficult to understand, we want to annotate each state with the sub-formulas of ϕ that are true in that state. For instance, Fig. 4 gives an explanation for why q_0 satisfies $\langle\!\langle transmitter \rangle\!\rangle\mathbf{G}\ \neg sent$. The explanation shows that, in q_0, the *block* action of the transmitter allows it to prevent the message to be sent.



$\langle *, block\rangle$
q_0
$\langle\!\langle transmitter \rangle\!\rangle\mathbf{G}\ \neg sent$
$\langle\!\langle transmitter \rangle\!\rangle\mathbf{X}\ \langle\!\langle transmitter \rangle\!\rangle\mathbf{G}\ \neg sent$
$\neg sent$

Fig. 4. An explanation for why the transmitter can prevent the value to be sent.

3.2 μ-Calculus Explanations

A μ-calculus explanation is a graph where nodes are triplets—called *obligations*—composed of a state q of S, a μ-calculus formula ϕ, and an environment e. An edge $\langle\langle q,\phi,e\rangle,\langle q',\phi',e'\rangle\rangle$ encodes the fact that $q \in [\![\phi]\!]^S e$ *because* $q' \in [\![\phi']\!]^S e'$. In this section, all μ-calculus formulas are considered in *positive normal form*, that is, all negations are applied to atomic propositions or variables only.

More formally, given a Kripke structure $S = \langle Q, \{R_i \mid i \in \Sigma\}, V\rangle$, an explanation is a graph $E = \langle O, T\rangle$ such that the nodes $O \subseteq Q \times \mathcal{L}_\mu \times \mathcal{E}$ are triplets of states of S, μ-formulas and environments, and the edges $T \subseteq O \times O$ link obligations together. The set of successors of o is noted $succ(o) = \{o' \mid \langle o, o'\rangle \in T\}$.

We are interested in explanations that are *adequate*, that is, that effectively show why q satisfies ϕ in environment e. An explanation E is adequate for explaining why $q \in [\![\phi]\!]^S e$ if it is *consistent*, *matches* S—that is, is composed of elements of S—and $\langle q,\phi,e\rangle \in O$.

An explanation is *consistent* if it exhibits the different parts needed to explain its elements. More formally, let $E = \langle O, T\rangle$ be an explanation and let $o = \langle q,\phi,e\rangle \in O$. o is said to be *locally consistent in E* iff

- $\phi \neq false$;
- if $\phi = true$, then $succ(o) = \varnothing$;
- if $\phi = p$ or $\phi = \neg p$, for $p \in AP$, then $succ(o) = \varnothing$;
- if $\phi = v$ or $\phi = \neg v$, for $v \in Var$, then $q \in e(v)$ (resp. $q \notin e(v)$) and $succ(o) = \varnothing$;
- if $\phi = \phi_1 \wedge \phi_2$ then $succ(o) = \{\langle q,\phi_1,e\rangle, \langle q,\phi_2,e\rangle\}$;

– if $\phi = \phi_1 \lor \phi_2$ then $succ(o) = \{\langle q, \phi_j, e\rangle\}$ for some $j \in \{1, 2\}$;
– if $\phi = \Diamond_i \phi'$ then $succ(o) = \{\langle q', \phi', e\rangle\}$ for some state q';
– if $\phi = \Box_i \phi'$ then, for all $o' \in succ(o)$, $o' = \langle q', \phi', e\rangle$ for some state q';
– if $\phi = \mu v.\psi(v)$, then $succ(o) = \{\langle q, \psi^k(\mathit{false}), e\rangle\}$ for some $k \geq 0$;
– if $\phi = \nu v.\psi(v)$, then $succ(o) = \{\langle q, \psi(\phi), e\rangle\}$.

E is then *consistent* iff all obligations $o \in O$ are locally consistent in E. Intuitively, if $\phi = \mu v.\ \psi$, then $q \in [\![\phi]\!]^S e$ because q belongs to a finite number of applications of ψ on *false*, that is, $q \in [\![\psi^k(\mathit{false})]\!]^S e$ for some $k \geq 0$. On the other hand, this idea cannot be applied for $\phi = \nu v.\ \psi$. In this case, $q \in [\![\phi]\!]^S e$ because it belongs to any number of applications of ψ on *true*. Thus, to explain it, E simply shows that $q \in [\![\psi(\phi)]\!]^S e$ and relies on the fact that the structure has a finite number of states to ensure that the explanation is finite as well.

Furthermore, E matches S iff

1. for all $\langle q', \phi', e'\rangle \in O$, $q' \in Q$;
2. for all $\langle q', p, e'\rangle \in O$, $p \in V(q')$ and for all $\langle q', \neg p, e'\rangle \in O$, $p \notin V(q')$;
3. for all $\langle\langle q', \phi', e'\rangle, \langle q'', \phi'', e''\rangle\rangle \in T$, either $q' = q''$, or ϕ' belongs to $\{\Diamond_i \phi'', \Box_i \phi''\}$ and $\langle q', q''\rangle \in R_i$;
4. for all $o' = \langle q', \Box_i \phi', e'\rangle \in O$, $\langle q', q''\rangle \in R_i$ iff $\exists o'' \in succ(o')$ s.t. $o'' = \langle q'', \phi'', e''\rangle$.

E matches S if E is part of S: (1) the states of E are states of S; (2) atomic propositions of E are coherent with labels of S; (3) successor states in E are successors in S; (4) the explanation for the $\Box_i$ operator exhibits all successors through R_i.

For instance, Figure 5 gives an adequate explanation for ϕ_{ns} (of Equation 1) holding in state q_0 of the µ-calculus structure of the bit transmission problem.

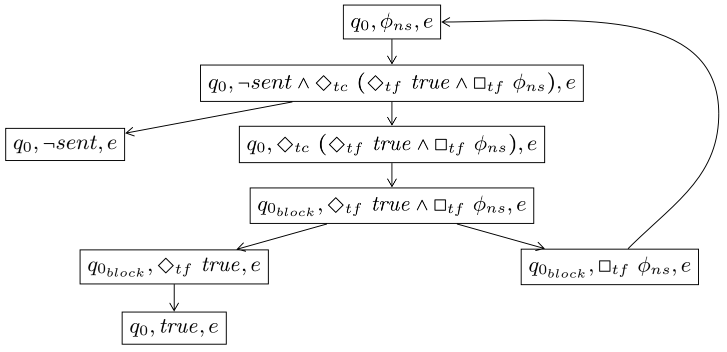

q_0, ϕ_{ns}, e
$q_0, \neg sent \land \Diamond_{tc}\ (\Diamond_{tf}\ true \land \Box_{tf}\ \phi_{ns}), e$
$q_0, \neg sent, e$
$q_0, \Diamond_{tc}\ (\Diamond_{tf}\ true \land \Box_{tf}\ \phi_{ns}), e$
$q0_{block}, \Diamond_{tf}\ true \land \Box_{tf}\ \phi_{ns}, e$
$q0_{block}, \Diamond_{tf}\ true, e$
$q0_{block}, \Box_{tf}\ \phi_{ns}, e$
$q_0, true, e$

Fig. 5. An explanation for why $q_0 \in [\![\phi_{ns}]\!]^S e$ in the bit transmission problem. tc and tf mean *transmitter chooses* and *transmitter follows*, respectively.

Adequate explanations are necessary and sufficient proofs for why $q \in [\![\phi]\!]^S e$, captured by the following property.

Property 1. Given a Kripke structure $S = \langle Q, \{R_i \mid i \in \Sigma\}, V\rangle$, a state $q \in Q$, a µ-calculus formula ϕ and an environment e, $q \in [\![\phi]\!]^S e$ if and only if there exists an adequate explanation E for $q \in [\![\phi]\!]^S e$.

Proof (Proof Sketch). The left-to-right direction is proved by the generating algorithm of this paper: if $q \in [\![\phi]\!]^S e$, then it generates an adequate explanation for $q \in [\![\phi]\!]^S e$. The other direction can be shown by induction over the structure of ϕ. The main idea is that, if E is adequate for sub-formulas, then local consistency and matching S are sufficient conditions for the formula to be satisfied. $\square$

Furthermore, we can view adequate explanations as patterns. An explanation E defines an entire set of Kripke structures $\mathcal{K}(E)$ that E matches. E is thus an explanation for why all structures of $\mathcal{K}(E)$ satisfy any formula ϕ that E contains. This intuition is formally captured by the following property.

Property 2. Given a consistent explanation $E = \langle O, T\rangle$, for all $\langle q, \phi, e\rangle \in O$, $q \in [\![\phi]\!]^S e$ for all S such that E matches S.

Proof. This property is directly derived from Property 1. If E is consistent, E matches S and $\langle q, \phi, e\rangle \in O$, then E is adequate for $q \in [\![\phi]\!]^S e$. By Property 1, since there exists an adequate explanation for $q \in [\![\phi]\!]^S e$, $q \in [\![\phi]\!]^S e$ is true. $\square$

Finally, we can define an algorithm to generate adequate explanations for µ-calculus formulas, presented in Algorithm 1. It takes a Kripke structure S, a state q of S, a µ-calculus formula ϕ, and an environment e such that $q \in [\![\phi]\!]^S e$, and returns an adequate explanation for $q \in [\![\phi]\!]^S e$. Intuitively, the algorithm starts with an empty explanation and the $\langle q, \phi, e\rangle$ obligation in the *pending* set. Then it considers each obligation $o' \in$ *pending*, adding to O and T the necessary obligations and edges to make o' locally consistent and matching S, and adding to *pending* the newly discovered obligations. It stops the process when all obligations have been made locally consistent in $\langle O, T\rangle$.

3.3 Translating µ-Calculus Explanations

The previous section proposed a structure to explain why a µ-calculus formula is satisfied by a state of some Kripke structure. Nevertheless, as the µ-calculus model checker and explanations are used to solve the model-checking problem of some other top-level logic, the usefulness of such explanations is limited. This section presents the set of functionalities the framework provides to help the designer to translate the µ-calculus explanations back into the top-level logic. They are generic to allow her to easily translate the explanations for logics such as CTL, CTLK, ATL or PDL [16], as well as fair variants such as Fair CTL [13].

First, aliases allow the designer to hide µ-calculus translations behind top-level logic formulas. Second, to ease the translation of explanations back into the original model language, the framework integrates the relational graph algebra of Dong et al. [15]. This algebra allows the designer to translate the explanation back into the original model language, but it treats the explanation as a whole.

Algorithm 1: $explain(S, q, \phi, e)$

```
Data: $S = \langle Q, \{R_i \mid i \in \Sigma\}, V\rangle$ a Kripke structure, $q \in Q$ a state of $S$, $\phi$ a
      µ-calculus formula, and $e$ an environment such that $q \in [\![\phi]\!]^S e$.
Result: An adequate explanation for $q \in [\![\phi]\!]^S e$.
$O = \varnothing$; $T = \varnothing$; $pending = \{\langle q, \phi, e\rangle\}$
while $pending \neq \varnothing$ do
    pick $o' = \langle q', \phi', e'\rangle \in pending$
    $pending = pending \setminus \{o'\}$
    $O = O \cup \{o'\}$
    case $\phi' \in \{true, p, \neg p, v, \neg v\}$: $O' = \varnothing$
    case $\phi' = \phi_1 \wedge \phi_2$: $O' = \{\langle q', \phi_1, e'\rangle, \langle q', \phi_2, e'\rangle\}$
    case $\phi' = \phi_1 \vee \phi_2$
        if $q' \in [\![\phi_1]\!]^S e'$ then $O' = \{\langle q', \phi_1, e'\rangle\}$ else $O' = \{\langle q', \phi_2, e'\rangle\}$
    case $\phi' = \Diamond_i\ \phi''$
        pick $q'' \in \{q'' \in Q \mid \langle q', q''\rangle \in R_i \wedge q'' \in [\![\phi'']\!]^S e'\}$
        $O' = \{\langle q'', \phi'', e'\rangle\}$
    case $\phi' = \Box_i\ \phi''$: $O' = \{\langle q'', \phi'', e'\rangle \mid \langle q', q''\rangle \in R_i\}$
    case $\phi' = \mu v.\ \psi$
        $\phi'' = false$; $sat = [\![\phi'']\!]^S e'$
        while $q' \notin sat$ do
            $\phi'' = \psi(\phi'')$; $sat = [\![\phi'']\!]^S e'$
        $O' = \{\langle q', \phi'', e'\rangle\}$
    case $\phi' = \nu v.\ \psi$: $O' = \{\langle q', \psi(\phi'), e'\rangle\}$
    $T = T \cup \{\langle o', o''\rangle \mid o'' \in O'\}$
    $pending = pending \cup (O' \setminus O)$
return $\langle O, T\rangle$
```

To ease the addition of information to individual obligations and edges, the framework also provides the notion of attributors. Finally, local translators are proposed to treat small parts of the given graph.

These functionalities help the designer to translate the µ-calculus explanation into another graph that is closer to the initial model language. Nevertheless, the designer has no control on the initial explanation the algorithm produces. To allow the designer to interfere into the choices the *explain* algorithm makes, the framework provides *choosers*.

Aliases An alias α is a syntactic function that takes a set of arguments and returns an *aliased* µ-calculus formula. The alias of an aliased formula is then used to hide the latter behind something more intelligible. For instance, the alias $\langle\!\langle\rangle\!\rangle\mathbf{X}(\Gamma, \phi) = \Diamond_{\Gamma choose}\ (\Diamond_{\Gamma follow}\ true \wedge \Box_{\Gamma follow}\ \phi)$ replaces the formula ϕ_{ns} with $\nu v.\ \neg sent \wedge \langle\!\langle transmitter\rangle\!\rangle\mathbf{X}\ v$.

Relational Graph Algebra The relational graph algebra of Dong et al. includes operators such as the union $G_1 \cup G_2$ and intersection $G_1 \cap G_2$ of two graphs G_1 and G_2, the selection $\sigma_{f_v,f_e}(G)$ of nodes and edges satisfying a condition, the projection $\pi_{d_v,d_e}(G)$ of nodes and edges on sub-domains d_v and d_e, the grouping $\gamma_{d_v,d_e}(G)$ of nodes and edges, etc. Thanks to this algebra, the designer can transform explanations into other graphs.

Obligation and Edge Attributors An *attribute* is data associated to explanation nodes and edges, and an *attributor* is a function adding attributes to an obligation or edge. They work as local decorators, in the sense that they deal with obligations and edges one at a time. They can be given to the generating algorithm to be run on every obligation or edge, or they can be attached to individual aliases to be run only on the obligations with instantiations of the aliases, or outgoing edges of these obligations. This improves the performances of decorating the graph when only a few elements must be decorated. In the case of ATL, we can define an attributor to attach to obligations the original CGS state their state derives from.

Local Translation A local translator is a function taking a relational graph and a particular node as arguments, and updating the graph. The part of the explanation a local translator receives is defined by the alias it is attached to. For instance, with a local translator, we can add edges to an explanation between an obligation labelled with a $\langle\!\langle\rangle\!\rangle\mathbf{X}$ alias and all the original successors of its state. The advantage of such a local translator is that the part of the graph it receives is the one explaining the $\langle\!\langle\rangle\!\rangle\mathbf{X}$ operator only.

Choosers and Partial Explanations A chooser takes an obligation and a set of possible successors of this obligation and returns a subset of these successors depending on the operator of the formula of the given obligation:

- for $\vee$ and $\Diamond_i$ operators, at most *one* successor must be chosen, to ensure a consistent explanation.
- for $\wedge$ and $\Box_i$ operators, the full explanation shows all successors, but a subset can be returned.
- for the other operators, there is no meaningful choice: there is no successor for *true* formulas, atomic propositions or variables, and there is only one successor for least and greatest fixpoint formulas.

Choosers can guide the explanation generation by choosing particular successors, but also limit the size of the generated explanation by only exploring parts of it. This introduces the notion of *partial explanations*, that is, explanations where some obligations are not fully explained because they lack some successors. The advantage of partial explanations is that the complete explanation can be too large to be generated or understood, so getting a part of it is better than nothing. Furthermore, choosers enable interactive generation of explanations as they can ask the user to resolve some choices.

Markers They are attached to formulas. The framework provides two types of markers, *points of interest*, and *points of decision*, but new types can be defined by the designer. Points of interest are intended to mark the formulas that are important for the designer. On the other hand, the model checker takes points of decision into account when generating explanations: whenever an obligation formula is marked with such a point, the model checker does not explain it. This produces partial explanations that can be later expanded by the user by forcing the generation of the missing parts.

Thanks to all these features, it is possible to transform the µ-calculus explanation of Fig. 5 for the formula $\langle\langle transmitter \rangle\rangle \mathbf{G}\ \neg sent$ and get the explanation of Fig. 6. For this translation, we used:

- aliases to hide µ-calculus formulas behind their ATL counterparts,
- points of interest for marking the formulas that have an ATL counterpart,
- an obligation attributor to extend each obligation with the original state,
- a local translator to add the edge with the action of the transmitter,
- the relational graph algebra to merge nodes together and gather the formulas that the state satisfies.

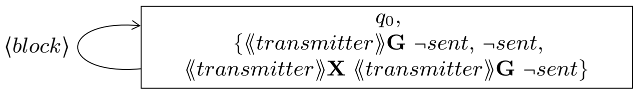

$\langle block \rangle$
q_0,
$\{\langle\langle transmitter \rangle\rangle \mathbf{G}\ \neg sent, \neg sent,$
$\langle\langle transmitter \rangle\rangle \mathbf{X}\ \langle\langle transmitter \rangle\rangle \mathbf{G}\ \neg sent\}$

Fig. 6. A translation of the µ-calculus explanation of Fig. 5 using the translation features of the framework.

4 Implementation

The framework has been implemented in Python using PyNuSMV for solving the model-checking problem. PyNuSMV is a library for prototyping symbolic model-checking algorithms based on NuSMV [6]. The implementation and examples are available on `http://lvl.info.ucl.ac.be/FM2018/FM2018`.

First, to be able to use the framework, the designer has to derive, from the original model, a µ-calculus Kripke structure $S = \langle Q, \{R_i \mid i \in \Sigma\}, V \rangle$. Such a structure is implemented with PyNuSMV as a standard SMV model to which several transition relations R_i are attached.

Second, the framework provides Python classes to define µ-calculus formulas, one for each µ-calculus operator: `MTrue`, `MFalse`, `Atom`, `Variable`, `Not`, `And`, `Or`, `Diamond`, `Box`, `Mu`, and `Nu`. With this implementation, µ-calculus formulas do not have to be declared in positive normal form. Instead, the framework lazily derives positive normal forms when needed. This allows the formulas that annotate the obligations to stay as close to the main formula as possible.

Third, most of the features are implemented with Python *decorators*, that is, function annotations that change the function behavior. For instance, aliases are defined as Python functions returning the corresponding µ-calculus formula and decorated with the `@alias` decorator. The code of Figure 7 shows a small part of the ATL model checker built with the framework. The `CAX` function returns the translation of the $[\![$`agents`$]\!]\mathbf{X}$ `formula` formula, marked with points of interest and decision, and to which is attached the `chosen_action` edge attributor.

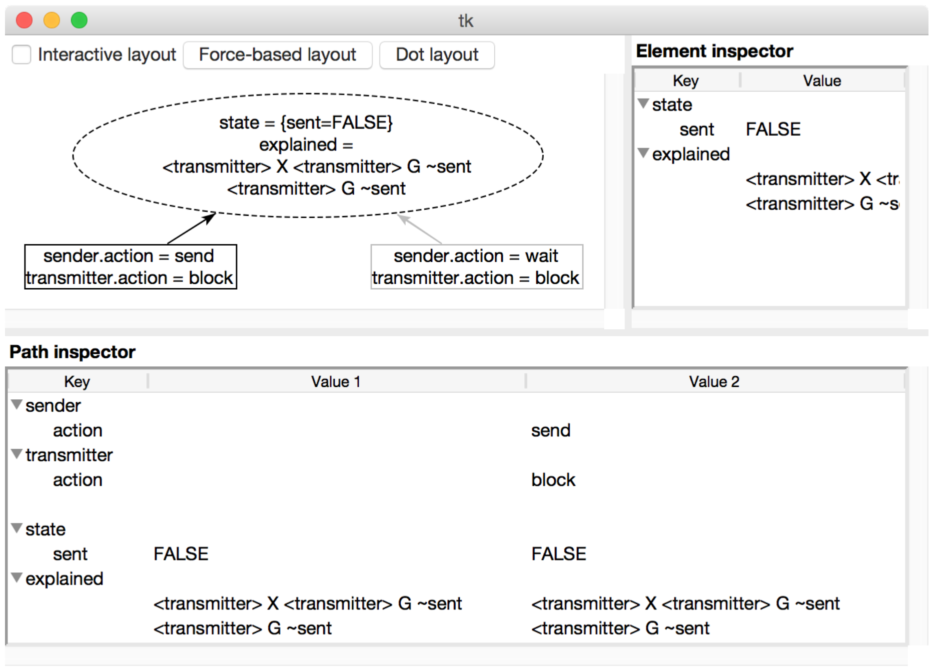
```
@alias("[{agents}] X {formula}")
def CAX(agents, formula):
    return POD(POI(chosen_action(
           Box(agents + "_choose",
               Or(Box(agents + "_follow", MFalse()),
                  Diamond(agents + "_follow", formula)))
          )))
@edge_attributor
def chosen_action(edge):
    # ...
    return {"action": actions}
```

Fig. 7. A part of the implementation of the ATL model checker built with the framework.

Relational graphs, and generated explanations in particular, are implemented with the `Graph` class. Nodes and edges of these graphs are implemented with the `domaintuple` class, a dictionary-like structure where domains of the elements are identified by a name. Each operator of the relational graph algebra is implemented by a method of the `Graph` class.

The framework allows the designer to efficiently translate an explanation back into the top-level language. Nevertheless, these explanations remain complex and difficult to understand. To help the user in understanding these complex explanations, the implementation also provides a graphical visualization tool. A snapshot of the tool is given in Figure 8.

The top left part presents the explanation: nodes are depicted in ovals, and edges are depicted as arrows decorated with information in a box. This graph can be moved with the mouse or automatically re-arranged. The information displayed in nodes and edge labels come from the explanation elements themselves. The tool also allows the user to select which keys of the graph elements are displayed, through a right-click menu on the graph area. To enable interactivity, the designer can specify a graphical menu that is displayed whenever the user right-clicks on the element. This can be used, for instance, to expand partial explanations. The top right part of the tool displays the complete information of the selected element (the dashed one on Fig. 8). The bottom part of the tool can display one particular path of the graph, selected by the user.



Element inspector

Key	Value
▼ state	
sent	FALSE
▼ explained	
	<transmitter> X <tr
	<transmitter> G ~s

Path inspector

Key	Value 1	Value 2
▼ sender		
action		send
▼ transmitter		
action		block
▼ state		
sent	FALSE	FALSE
▼ explained		
	<transmitter> X <transmitter> G ~sent	<transmitter> X <transmitter> G ~sent
	<transmitter> G ~sent	<transmitter> G ~sent

Fig. 8. A snapshot of the visualization tool.

5 Application to ATL

The objective of this section is to show the usefulness of the framework by applying it to the ATL logic. It describes how explanations for ATL can be obtained, displayed and manipulated thanks to the framework implementation.

The implementation represents a CGS with a standard SMV model to which is attached a set of agents. Each agent has a name and a set of SMV input variables corresponding to its actions. The SMV model itself defines what the agents can do, and how the state of the model evolves according to their actions.

The translation of the CGS acts like a dictionary of transition relations, lazily building these relations when needed. The advantage of this mechanism is that, even if the CGS contains a lot of agents, its implementation builds the transition relations only for the groups of agents appearing in the checked formula. The translation of ATL formulas simply uses the Python classes provided by the framework to define μ-calculus formulas.

To enrich and translate explanations, one alias is declared for each ATL operator. All top-level formulas returned by the aliases are marked as points of interest. Furthermore, both $\langle\!\langle\rangle\!\rangle\mathbf{X}$ and $[\![]\!]\mathbf{X}$ aliases are marked as points of decision, to be able to generate small partial explanations and to allow the user to expand them as she wishes.

Two attributors add information to obligations and edges of the explanation. The first attributor attaches, to each obligation, the original state its state derives from. This attributor is given to the *explain* algorithm to enrich all obligations. The second attributor stores the actions chosen by the group in the outgoing edge of the obligations labelled with a $\langle\langle\rangle\rangle\mathbf{X}$ or $[[]]\mathbf{X}$ aliased formula. This way, the information is more easily accessed by local translators. Figure 7 illustrates these parts of the implementation.

Two local translators are defined, for $\langle\langle\rangle\rangle\mathbf{X}$ and $[[]]\mathbf{X}$. They extract, from the two steps of the µ-calculus model, the original one-step transitions of the CGS. The relational graph algebra is used to translate µ-calculus explanations back into ATL ones. The translation:

1. projects the explanation nodes on formulas and original states;
2. groups nodes by their original state;
3. separates unexplained formulas from explained ones;
4. selects edges that are labelled with some actions;
5. keeps the original state and the formulas in nodes, and the actions in edges.

This translation produces explanations such as the one of Figures 6 and 8.

A chooser is defined to expand partial explanations. When dealing with a $\langle\langle\rangle\rangle\mathbf{X}$ alias, it gets the original actions of the group from the given successors and asks the user to choose one of them through a window. Finally, the visualization tool is used to display and manipulate the translated explanations, as shown in Fig. 8. In particular, it provides, through a right-click menu, the list of unexplained formulas. This menu triggers the expansion of the currently displayed partial explanation, running through the chooser to select the action to play.

6 Related Work

Several authors already proposed solutions to explain why a CTL formula is satisfied by some model. First, some authors proposed structures capturing the part of the model witnessing the satisfaction [29,11,3]. These structures are defined as hierarchies of paths, fitting the CTL semantics. Jiang and Ciardo recently proposed a way to generate such hierarchies of paths with a *minimal* number of states [19]. Other authors proposed more detailed structures, capturing the part of the model, as well as sub-formulas and logical decomposition steps [30,18,7,8,31,27,33]. These different solutions vary in terms of details they provide about the satisfaction—by annotating or not the parts of the counter-example with the sub-formulas they explain—, the fragments of the logic they support—either the full logic or its universal fragment—, and the framework they work in—explicit, game-based, proof-based, BDD-based model checking, or Boolean Equation Systems (BES). All these solutions can be adapted to a BDD-based framework and produced with the framework we propose.

Some solutions have also been proposed in the context of multi-modal logics, adapting and extending the ideas from CTL to richer logics [26,24,25,5,34]. In this context, MCK, a tool for verification of temporal and knowledge properties, provides several debugging functionalities [17], such as a debugging game inspired by Stirling's games [32] in which the user can try to show why the model-checking outcome is wrong while the system shows her why it is actually right. Such a debugging game can be implemented with adequate choosers.

Finally, several solutions have also been proposed to represent and produce explanations for the μ-calculus [21,20,23,14]. They differ from the ones presented in this paper either by the way they are generated—such as the explanations of Kick [21]—or by the actual framework they rely on.

All these solutions work for particular logics such as CTL, CTLK, the μ-calculus, or are generic solutions with some application to one use case such as BES and their extensions, games, or proofs. But no work proposes a solution to produce explanations and to translate them back into the original language, as the μ-calculus framework of this paper. They either limit themselves to one logic, or they provide generic structures without giving explicit help for applying and translating it into something useful for the end user.

7 Conclusion

In this paper, we described a solution for μ-calculus-based logics explanations. The proposed framework integrates a μ-calculus model checker that generates rich explanations and provides several functionalities to translate them into explanations for a top-level logic such as ATL. It has been implemented with PyNuSMV, taking advantage of Python functionalities such as function decorators to easily describe the different features. The implementation also integrates a graphical tool to visualize, manipulate and explore the explanations.

One of the main advantages of the framework is that many logics can be translated into the μ-calculus, such as CTL, Fair CTL, CTLK, ATL, and PDL. It is thus generic enough to provide model-checking functionalities for all of them. Furthermore, thanks to the framework, the designer does not have to worry about designing and implementing a model checker, nor about generating rich explanations. Nevertheless, she has to translate the top-level models and formulas into μ-calculus. Model translation can be difficult—for instance, the translation from an ATL CGS to a μ-calculus structure is not trivial—and the framework gives no help to complete this task.

The framework features allow the designer to divide the concerns into smaller parts, first dealing with formula translations (with aliases and markers), then with single elements (with attributors), small sub-graphs (with local translation), and with the whole explanation (with the algebra). Furthermore, all the features are useful, as illustrated by the ATL case. In particular, local translators are useless for cases such as CTL, but for ATL, where the model translation is difficult, they can help treating small parts of the explanation separately, instead of having to deal with the whole explanation graph at once. The visualization tool provided by the framework complements the translation features. The latter help the designer to produce useful explanations while the former helps the user visualize, manipulate and explore it.

Finally, the framework supports interactive and guided generation of the explanations through choosers. This can lead to smaller manageable partial explanations that can be interactively expanded, as illustrated by the ATL case.

One of the main drawbacks of the framework is the fact that it produces one single explanation at a time. Representing several explanations at once could help the user to extract the reasons for the satisfaction of the formula more easily. As future work, it would be interesting to explore how we could represent several explanations at once by using binary decision diagrams to represent *sets of obligations* instead of single ones. Furthermore, translating a CGS and an ATL formula into a μ-calculus model and a formula is not an easy task compared to other logics such as CTL and CTLK. One solution to make this particular translation easier is to use the alternating-time μ-calculus [1] as base logic instead of the propositional μ-calculus. Finally, it would be interesting to explore solutions to provide translation functionalities for the model itself. With such translation functionalities, the translation of explanations back into the original language could become automatic.

References

1. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM 49(5), 672–713 (Sep 2002)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
3. Buccafurri, F., Eiter, T., Gottlob, G., Leone, N.: On ACTL formulas having linear counterexamples. Journal of Computer and System Sciences 62(3), 463 – 515 (2001)
4. Busard, S.: Symbolic model checking of multi-modal logics: uniform strategies and rich explanations. Ph.D. thesis, Université catholique de Louvain (July 2017)
5. Busard, S., Pecheur, C.: Rich counter-examples for temporal-epistemic logic model checking. In: Proceedings Second International Workshop on Interactions, Games and Protocols, IWIGP 2012, Tallinn, Estonia, 25th March 2012. pp. 39–53 (2012), `http://dx.doi.org/10.4204/EPTCS.78.4`
6. Busard, S., Pecheur, C.: PyNuSMV: NuSMV as a Python library. In: Brat, G., Rungta, N., Venet, A. (eds.) Nasa Formal Methods 2013. LNCS, vol. 7871, pp. 453–458. Springer-Verlag (2013)
7. Chechik, M., Gurfinkel, A.: A framework for counterexample generation and exploration. In: International Conference on Fundamental Approaches to Software Engineering. pp. 220–236. Springer (2005)
8. Chechik, M., Gurfinkel, A.: A framework for counterexample generation and exploration. International Journal on Software Tools for Technology Transfer 9(5-6), 429–445 (2007)
9. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer Berlin Heidelberg (2002), `http://dx.doi.org/10.1007/3-540-45657-0_29`
10. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
11. Clarke, E.M., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: Proc. of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002). pp. 19–29 (2002)

12. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logics of Programs, Workshop, Yorktown Heights, New York, May 1981. pp. 52–71 (1981), `http://dx.doi.org/10.1007/BFb0025774`
13. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986), `http://doi.acm.org/10.1145/5397.5399`
14. Cranen, S., Luttik, B., Willemse, T.A.: Proof graphs for parameterised boolean equation systems. In: International Conference on Concurrency Theory. pp. 470–484. Springer (2013)
15. Dong, Y., Ramakrishnan, C.R., Smolka, S.A.: Model checking and evidence exploration. In: Proc. of the 10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003). pp. 214–223 (2003)
16. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. Journal of computer and system sciences 18(2), 194–211 (1979)
17. Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Proceedings of 16th International Conference on Computer Aided Verification (CAV'04). LNCS, vol. 3114, pp. 479–483. Springer-Verlag (2004)
18. Gurfinkel, A., Chechik, M.: Proof-like counter-examples. In: Garavel, H., Hatcliff, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 2619, pp. 160–175. Springer Berlin / Heidelberg (2003)
19. Jiang, C., Ciardo, G.: Generation of minimum tree-like witnesses for existential CTL. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 328–343. Springer International Publishing, Cham (2018)
20. Kick, A.: Generation of witnesses for global μ-calculus model checking. Tech. rep., Universität Karlsruhe, Germany (1995)
21. Kick, A.: Tableaux and witnesses for the μ-calculus. Tech. rep., Universität Karlsruhe, Germany (1995)
22. Kozen, D.: Results on the propositional mu-calculus. Theor. Comput. Sci. 27, 333–354 (1983), `http://dx.doi.org/10.1016/0304-3975(82)90125-6`
23. Linssen, C.A.: Diagnostics for Model Checking. Master's thesis, Eindhoven University of Technology (2011)
24. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A model checker for the verification of multi-agent systems. In: Proceedings of CAV 2009. LNCS, vol. 5643, pp. 682–688. Springer (2009)
25. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. International Journal on Software Tools for Technology Transfer pp. 1–22 (2015), `http://dx.doi.org/10.1007/s10009-015-0378-x`
26. Lomuscio, A., Raimondi, F.: MCMAS: A model checker for multi-agent systems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 450–454. Springer (2006)
27. Mateescu, R.: Efficient diagnostic generation for boolean equation systems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 251–265. Springer (2000)
28. Penczek, W., Lomuscio, A.: Verifying epistemic properties of multi-agent systems via bounded model checking. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems. pp. 209–216. AAMAS '03, ACM, New York, NY, USA (2003), `http://doi.acm.org/10.1145/860575.860609`

29. Rasse, A.: Error diagnosis in finite communicating systems. In: Larsen, K., Skou, A. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 575, pp. 114–124. Springer Berlin / Heidelberg (1992)
30. Roychoudhury, A., Ramakrishnan, C., Ramakrishnan, I.: Justifying proofs using memo tables. In: International Conference on Principles and Practice of Declarative Programming: Proceedings of the 2 nd ACM SIGPLAN international conference on Principles and practice of declarative programming. pp. 178–189 (2000)
31. Shoham, S., Grumberg, O.: A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. ACM Transactions on Computational Logic (TOCL) 9(1), 1 (2007)
32. Stirling, C.: Local model checking games. In: International Conference on Concurrency Theory. pp. 1–11. Springer (1995)
33. Tan, L., Cleaveland, R.: Evidence-based model checking. In: International Conference on Computer Aided Verification. pp. 455–470. Springer (2002)
34. Weitl, F., Nakajima, S., Freitag, B.: Structured counterexamples for the temporal description logic ALCCTL. In: 2010 8th IEEE International Conference on Software Engineering and Formal Methods. pp. 232–243. IEEE (2010)