Verifying social expectations by model checking truncated paths

Stephen Cranefield

Department of Information Science,
University of Otago,
PO Box 56, Dunedin 9054, New Zealand
Tel.: +64 3 479 8083
Fax: +64 3 479 8311

Michael Winikoff

Department of Information Science,
University of Otago,
PO Box 56, Dunedin 9054, New Zealand
Tel.: +64 3 479 8386
Fax: +64 3 479 8311

Abstract

One approach to moderating the expected behaviour of agents in open societies is the use of explicit languages for defining norms, conditional commitments and/or social expectations, together with infrastructure supporting conformance checking. This paper presents a logical account of the fulfilment and violation of social expectations modelled as conditional rules over a hybrid linear propositional temporal logic. Our semantics captures the intuition that the fulfilment or violation of an expectation must be determined without recourse to information from later states. We define a means of updating expectations from one state to the next based on formula progression, and show how conformance checking was implemented by combining the MCFULL model checking algorithm of Franceschet and de Rijke and the semantics for LTL over truncated paths proposed by Eisner *et al.* We present algorithms for both traditional *offline* model checking, where the complete model is available at once, and *online* model checking, where states are added to the model sequentially at run-time.

Email addresses: Stephen.Cranefield@otago.ac.nz (Stephen Cranefield), Michael.Winikoff@otago.ac.nz (Michael Winikoff)

Key words: Social expectations, model checking, multi-agent systems.

1. Introduction

An *electronic institution* [1] is an explicit model of the rules, or norms, that govern the operation of an open multi-agent system. A given electronic institution provides rules that agents participating in the institution are expected to follow. These rules can include more traditional protocols (e.g. a request message comes first, followed by either an accept or a refuse), as well as properties that are expected to apply to complete interactions, for example, the norm that any accepted request must be eventually fulfilled.

Since electronic institutions are open systems it is not possible to assume any control over agents, nor is it reasonable to assume that all agents will follow the rules applying to an interaction. Instead, the behaviour of participating agents needs to be monitored and checked, with violations being detected and responded to in a suitable way, such as punishing the agent by applying sanctions, or reducing the agent's reputation.

There is therefore a need for mechanisms to check for the fulfilment or violation of norms with respect to a (possibly partial) execution trace. Furthermore, such a mechanism can also be useful for rules of social interaction that are less authoritative than centrally established norms, e.g. conditional rules of expectation that an agent has established as its personal norms, or rules expressing learned regularities in the patterns of other agents' behaviour.

Thus in this paper we focus on modelling the general concept of *social expectation* and demonstrate the use of *model checking* for detecting the fulfilment or violation of such expectations by extending the MCFULL algorithm of the Hybrid Logics Model Checker [2]. The advantages of building on model checking, rather than implementing our own checking algorithm (as was done previously [3]) are that we are working within a clearly defined and well studied verification framework. Although the problem of model checking, in its full generality, is more complex than we need, the problem of model checking a path (a finite or ultimately periodic sequence of states) has also been studied and "can usually be solved efficiently, and profit from specialized algorithms" [4]. We have therefore investigated the applicability of model checking as a way of checking for expectations, fulfilments and violations over a model that is a linear history of observed states.

The theory underlying our approach is designed to apply equally well to both

online and *offline*[1] monitoring of expectations, a distinction that has not been made in previous work. For online monitoring, each state is added to the end of the history as it occurs, and the monitoring algorithm works incrementally. The underlying formalism can assume that expectations are always considered at the last state in the history. In contrast, in the offline mode, expectations in previous states are also checked. At each past state, the then-active expectations must be checked for fulfilment without recourse to information from later states: the *truth* of a future-oriented temporal proposition ϕ at state s over the full history does not imply the *fulfilment* at s of an expectation with content ϕ.

This paper is an expanded version of an earlier paper [5][2]. In addition to a more complete discussion of our model checking procedure and additional discussion of key points, we include an extended logic, a complexity analysis of our algorithm, two new examples (one more complex, and one illustrating new features of the logic in the context of Second Life), a detailed discussion of online checking, and extended discussion of related work.

This paper is structured as follows. Section 2 outlines our intuitions about expectations, fulfilment and violation, and motivates our logical account of these concepts. Section 3 describes the logic and the semantic mechanisms we use to express fulfilment and violation of an expectation. In Section 4 we give a brief description of formula progression, a technique used to express the evolution of an unfulfilled and non-violated expectation from one state to the next. Section 5 then describes the Hybrid Logics Model Checker that we have used in this work and the extensions we have made to it. Example scenarios in the domains of travel booking and monitoring virtual soccer training in Second Life are presented in Section 6. Section 7 discusses online checking in more detail. Finally we discuss related work in Section 8 and summarise the paper and plans for future work in Section 9.

2. Formalising Expectations, Fulfilment and Violation

In this work we study the general notion of *expectations*. It is our position that the base-level semantics of expectations with different degrees of force (expectations inferred from experience, promises, formal commitments, etc.) are the

[1] Offline monitoring can be implemented by applying an online algorithm iteratively, but this is not the most natural approach theoretically nor the most efficient approach in practice.

[2] Text and figures (numbers 1, 6, 7 and 8) from the earlier paper [5] are included here with kind permission of Springer Science+Business Media.

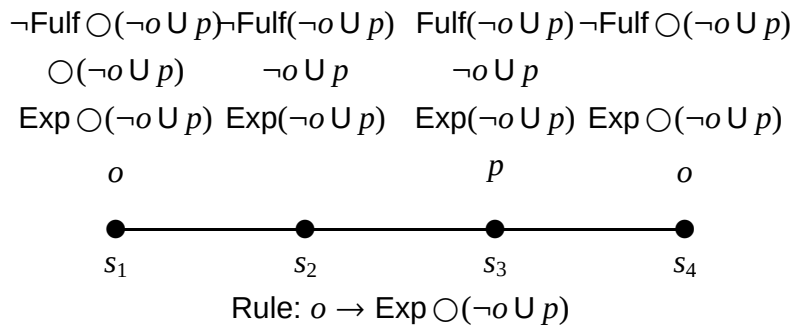


Figure 1: An example rule and scenario

same. The differences between these lie in the pragmatics of how they are created and propagated, how their fulfilment and violation is handled, and the type of contextual information associated with them (e.g. the debtor and creditors associated with a commitment). While there are many important research problems in the modelling and reasoning about norms and commitments, in this research we focus on the *temporal* aspects of expectations and their fulfilment and violation.

Our intuition behind expectations is that they are created in some context which may depend on the current and recorded past states of an agent (including any representation it has of the external environment), and that the created expectation is a constraint that should be applied to the future execution trace. We model this by conditional rules:

$$\lambda \rightarrow \mathsf{Exp}\,\rho$$

where λ and ρ are linear temporal logic (LTL) expressions with λ referring to the past and present and ρ representing the constraint. Note that whilst we require that λ should refer to the past and present only, ρ can express a constraint on the future to be monitored or a property of the history to be checked (see Section 6). The modality Exp is needed as it is not *guaranteed* that ρ will hold; it will just be *expected* if the condition holds.

The question then arises of when an expectation should be considered to be fulfilled (denoted $\mathsf{Fulf}\,\phi$) or violated ($\mathsf{Viol}\,\phi$). Consider Figure 1. This shows a rule expressing an expectation on the interaction between a merchant and a customer: a customer agent that has placed an order (modelled as the proposition o) should not subsequently place another order until its order has been paid for (proposition p). We formalise this as $o \rightarrow \mathsf{Exp}\,\bigcirc(\neg o\,\mathsf{U}\,p)$, i.e., when o holds, it is expected that, from the next state on, o is false until p holds. In the figure, the bottom

row of formulae show the propositions that are observed in a segment of one possible history: o holds in states s_1 and s_4. and p holds in s_3. The row above this shows the expectations that are *created* by the rule (in states s_1 and s_4) and then *updated* from one state to the next (in states s_2 and s_3, using a technique discussed below). Above this we show the content formulae of these expectations for the first three states, which can easily be seen to hold due to the semantics of the temporal operators $\bigcirc$ and U (a longer segment is needed to evaluate the content of the expectation in s_4). However, as indicated in the top row, we should not necessarily conclude that these expectations are fulfilled just because their content formulae are true. The determination of fulfilment and violation must be made *without recourse to future information*. Thus, only in state s_3, when payment is made (p holds), should it be concluded that the current expectation is fulfilled. Section 3 presents a temporal operator $\mathsf{Trunc_S}$ that allows us to express this restriction to past and present information. For now, we will assume that we have suitable definitions of $\mathsf{Fulf}\,\phi$ and $\mathsf{Viol}\,\phi$, and move on to consider how expectations evolve from one state to the next.

We assume that, logically, an expectation can be fulfilled or violated at most once[3] and that an expectation that is not fulfilled or violated in a state should persist (in a possibly modified form) to the next state:

$$\mathsf{Exp}\,\phi \wedge \neg\,\mathsf{Fulf}\,\phi \wedge \neg\,\mathsf{Viol}\,\phi \rightarrow \bigcirc\,\mathsf{Exp}\,\psi$$

What should ψ be? Although at least one alternative approach exists (see Section 8), we believe that the most intuitive representation of an expectation is for it to be expressed in terms of the current state. Thus ψ should represent a change of viewpoint of the constraint represented by the expectation ϕ from the current state to the next state. This can be seen in Figure 1 where $\mathsf{Exp}\bigcirc(\neg o\,\mathsf{U}\,p)$ from s_1 becomes $\mathsf{Exp}(\neg o\,\mathsf{U}\,p)$ in s_2 and then remains as $\mathsf{Exp}(\neg o\,\mathsf{U}\,p)$ in s_3 as p was not true in s_2.

The transformation of ϕ into ψ should also take into account any simplification of the expectation due to subformulae of ϕ that were true in the current state. Thus, an expectation $\mathsf{Exp}(p \wedge \bigcirc q)$ should become $\mathsf{Exp}\,q$ in the next state if p holds currently. This is precisely the notion of formula progression through a state [7].

[3] In practice, extra-logical support for notification policies [6] is needed to allow users to control whether they receive repeated notifications for "always" or "never" type rules, e.g. it may be desirable to learn about every violation of the rule that one should never commit murder, even though logically the truth value of this constraint is completely determined after the first violation.

$$\mathsf{A}\mathsf{G}^{+}\,\mathrm{done}(c,\ \mathrm{buy\_sub}(\mathit{publication}, p, \mathit{price}))\ \wedge\ [t,\ t + \mathsf{P1W}\,|\,\mathsf{Z})\ \rightarrow$$
$$\downarrow_{\mathsf{Z}}^{\mathsf{week}} \mathrm{w}.\mathsf{G}^{+}_{[\mathrm{w}+\,\mathsf{P1W}\,|\,\mathsf{Z},\ \mathrm{w}+\,\mathsf{P53W}\,|\,\mathsf{Z})}$$
$$\downarrow_{\mathsf{Z}}^{\mathsf{week}} \mathrm{cw}.\downarrow_{\mathsf{Z}}^{\mathsf{now}} \mathrm{n}.(\neg \mathsf{X}^{-}[\mathrm{cw}, \mathrm{n}]\ \rightarrow$$
$$\exists \mathrm{d}\ (\mathrm{date\_time\_to\_date}(\mathrm{cw}, \mathrm{d}) \wedge$$
$$\mathsf{F}^{+}_{[\mathrm{n},\ \mathrm{cw}+\,\mathsf{P1W}\,|\,\mathsf{Z})}\mathrm{done}(p,\ \mathrm{send}(c, \mathit{publication}, \mathrm{d}))))$$

Figure 2: An example rule in hyMITL$^{\pm}$ [6]

Formula progression (which will be explained in more detail in Section 4) allows us to complete our informal characterisation of the evolution of expectations through time:

$$\mathsf{Exp}\,\phi \wedge \neg\,\mathsf{Fulf}\,\phi \wedge \neg\,\mathsf{Viol}\,\phi \wedge \mathsf{Progress}(\phi, \psi) \rightarrow \bigcirc\,\mathsf{Exp}\,\psi$$

This conception of expectation, fulfilment and violation has been implemented in a previous progression-based system using a logic (hyMITL$^{\pm}$) combining future and past temporal operators with the guarded fragment of first order logic, binders and a form of nominal [3, 6]. The logic allows the expression of temporally rich conditional expectations such as the rule shown in Figure 2. This rule states that if client *c* has bought a subscription to *publication* from provider *p* for *price* and this happens within a week of time *t* (the price is only valid for a week), then at all times within the interval beginning a week after the start of the week that the payment is made (w) and ending immediately before 53 weeks after w, if the current state is the first within a given week (encoded as the constraint that the previous state wasn't between the start of the week and now, inclusive), then between now and the end of the week the provider will send the current edition of the publication. In other words, once the payment is made, the publication will be sent every week for 52 weeks.

hyMITL$^{\pm}$ is a temporal logic that includes unary temporal operators (including standard abbreviations) meaning *in the next/previous state* ($\mathsf{X}^{+}/\mathsf{X}^{-}$), *eventually in the future/past* ($\mathsf{F}^{+}/\mathsf{F}^{-}$), and *always in the future/past* ($\mathsf{G}^{+}/\mathsf{G}^{-}$), as well as binary *until* operators for the future and past directions ($\mathsf{U}^{+}/\mathsf{U}^{-}$) and a *for all possible future sequences of states* operator (A). The F, G and U operators are qualified by temporal intervals to constrain the states that must be considered when evaluating the argument (or the second argument in the case of $\mathsf{U}^{+}/\mathsf{U}^{-}$). The default interval (if one is omitted) is $(-\infty, +\infty)$. An interval may also appear on its own as a

formula to express the constraint that the current state is within that time period. There is a "current time" binding operator ($\downarrow$) that is qualified by a time unit (e.g. week) and a time zone, and binds the following variable to a term that names the current time in the specified time zone, rounded down to the beginning of the specified unit of time. Time interval end points may refer to relative time points (e.g. P1W) and are expressed in a notation based on ISO standard 8601.

Although the previously developed system used an expressive logical notation for rules of expectation, the detection of fulfilments and violations and the progression of expectations from one state to the next were handled algorithmically, and there was not a logical account of these notions. This paper provides such a logical account for a simpler *propositional* temporal logic in conjunction with an elaboration on the Exp, Fulf and Viol operators discussed above.

In particular, rather than modelling rules of expectations as implications of the form $\lambda \rightarrow \mathsf{Exp}\,\rho$, our logic includes formulae of the form $\mathsf{ExistsExp}(\lambda, \rho)$, $\mathsf{ExistsFulf}(\lambda, \rho)$ and $\mathsf{ExistsViol}(\lambda, \rho)$, with the intended meanings that an expectation currently exists, is fulfilled, or is violated (respectively) as a result of the rule with condition λ and expectation ρ having been triggered in the current or a previous state. Due to the use of formula progression, the formulae that are currently expected, fulfilled or violated as a result of the rule may differ from ρ, and these are defined by the semantics of ExistsExp, ExistsFulf and ExistsViol, generated by the model checker, and reported in its output.

It is our longer-term aim to extend our logical account of expectations and our model checker so that we can include many of the additional features of hyMITL$^{\pm}$.

3. Formal Background

The logic we use to model social expectations in this paper is a hybrid temporal logic that is an extension of the one implemented by the Hybrid Logics Model Checker (HLMC) [2], which is based on the algorithms of Franceschet and de Rijke [8]. First we present the hybrid temporal logic underlying our account of expectation, leaving a formal account of the ExistsExp, ExistsFulf and ExistsViol until Section 5.4.

Our language is described by the following grammar, where ψ is the language accepted by the model checker, and ϕ is the sublanguage that is permitted within ExistsExp, ExistsFulf and ExistsViol, i.e. we do not currently permit nesting of

expectation modalities.

$$
\begin{aligned}
\psi &::= \mathsf{ExistsExp}(\phi_1, \phi_2) \mid \mathsf{ExistsFulf}(\phi_1, \phi_2) \mid \mathsf{ExistsViol}(\phi_1, \phi_2) \mid \phi \\
\phi &::= p \mid p(s) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc\phi \mid \ominus\phi \mid \phi_1 \,\mathsf{U}\, \phi_2 \mid \phi_1 \,\mathsf{S}\, \phi_2 \mid \\
&\quad\; s \mid @_s\,\phi \mid \downarrow x.\phi \mid \exists_{p(x)}\,\phi \\
s &::= x \mid n
\end{aligned}
$$

where p is a proposition, $p(s)$ is a "state-referencing proposition" (discussed below), $\bigcirc$ is the standard temporal "next" operator, $\ominus$ is the standard temporal "previous" operator, U is the standard temporal "until", S ("since") is a backwards-looking version of until, and the remaining formula types are hybrid logic constructs discussed below. We assume the propositions include $\top$ (true) and $\bot$ (false), with their usual meanings, and define as abbreviations the Boolean connectives $\vee$ and $\rightarrow$, the derived temporal operators "eventually ϕ" ($\Diamond\phi \equiv \mathit{true}\,\mathsf{U}\,\phi$), and "always ϕ" ($\Box\phi \equiv \neg\Diamond\neg\phi$), and similar backwards-looking versions $\diamondminus\phi \equiv \mathit{true}\,\mathsf{S}\,\phi$ and $\boxminus\phi \equiv \neg\diamondminus\neg\phi$. In some literature (and in our previous work on hyMITL$^{\pm}$) $\Diamond$ is denoted by $\mathbf{F}$, $\Box$ by $\mathbf{G}$, $\diamondminus$ by $\mathbf{F}^{-}$ and $\boxminus$ by $\mathbf{G}^{-}$.

State-referencing propositions allow a directed relationship with another state to be expressed[4]. In this paper we assume that these propositions are always used to refer to past states[5]. Motivation for this construct is given in Section 6.2, where the proposition $\mathit{goal}(s)$ is used to express the fact that in a game of football a goal-scoring action has just completed after being initiated by a kick in the prior state s. In other words, scoring a goal is treated as an action that takes place over an interval of time and which can only be known to have occurred at the end of that interval.

Although including past operators in linear temporal logic does not add any expressive power [9], these do allow a more concise expression of many properties of practical interest [10], and so we choose to include them in our language. As we are interested in the problem of "model checking a path" (an observed sequence of states) [4] rather than a general Kripke structure, the difficulties of model checking LTL+Past [11] do not apply for our application, although this choice did complicate the labelling procedure described in Section 5.2

[4]These relationships could also be encoded in the model as additional (non-temporal) accessibility relations, with corresponding model operators included in the language. However, as we do not define traditional modal operators based on these relationships, we have chosen to treat these as a form of proposition that is associated with a specific state.

[5]The truncated path semantics of $p(s)$ and $\exists_{p(x)}\,\phi$ presented in Section 3 require further development in order to account for the case when $p(s)$ refers to the future.

The first four hybrid logic features in our logic are standard. We have so-called *state variables*, with typical element x, and nominals n. A nominal is a logical proposition that is true in exactly one state, i.e. the state "designated" by the nominal. The operator $@_s\,\phi$, where s is either a state variable or a nominal, shifts evaluation to the state s and can be read as "ϕ holds in state s". The operator $\downarrow x.\phi$ binds the state variable x to the current state and then proceeds to evaluation of ϕ. The final operator, $\exists_{p(x)}\,\phi$, is a guarded quantifier. This states that there is some state x for which $p(x)$ currently holds, and ϕ also holds for that value of x.

The use of nominals is important: we rely on each state having a unique label in order to define our Exp modality (see Section 5.4). However, these nominals can be generated automatically by the model checker, and there is no requirement for these to appear explicitly in models or the formula to be checked.

Although the account of the fulfilment and violation of social expectations presented in this paper does not *require* the use of the other hybrid logic features in our language, we include these in the language as they were already implemented in the Hybrid Logics Model Checker (HLMC), which we initially extended in this work. These features turn out to be useful in conjunction with state-referencing propositions, as illustrated in the Second Life soccer scenario presented in Section 6.2.

Note that the logic of Franceschet and de Rijke [8] also has an existential binder $\exists x\phi$ which we leave out. HLMC instead has a non-binding variant $\mathsf{E}\phi$ (ϕ is true in some state), but as our models are linear traces, this can be replaced with $@_\alpha\Diamond\phi$ (where α is a nominal that is true in the first state).

The formal semantics for our logic are given in Figure 3 with respect to a hybrid Kripke structure $\mathcal{M}$, which consists of an infinite sequence of states $\langle m_1, m_2 \ldots\rangle$ and a valuation function V that maps ordinary propositions and nominals to the set of states in which they hold, i.e. $\mathcal{M} = \langle\langle m_1, m_2 \ldots\rangle, V\rangle$. V is extended to map each state-referencing proposition p to a function that determines, for each state m_i, the (finite) set of argument states for which p holds in m_i, i.e. $V(p) : S_\mathcal{M} \to \wp(S_\mathcal{M})$, where $S_\mathcal{M}$ is the set of states in $\mathcal{M}$. The function g maps state variables x to states, and we write $g[x \mapsto m_i]$ to denote the function that maps x to m_i and otherwise behaves like g. We write $[V, g](s)$ as an abbreviation for either the value of $V(s)$ if s is a nominal or the singleton set $\{g(s)\}$ if s is a state variable.

Note that the rules of Figure 3 only apply for $i \geq 1$. For $i < 1$ we have $\mathcal{M}, g, i \not\models \phi$.

When evaluating whether an expectation is fulfilled in a state m_i we want to not only determine whether the formula holds, but also whether an agent in state

$$
\begin{array}{rll}
\mathcal{M}, g, i \models p & \text{iff} & m_i \in V(p) \\
\mathcal{M}, g, i \models p(s) & \text{iff} & [V, g](s) \in V(p)(m_i) \\
\mathcal{M}, g, i \models \neg\phi & \text{iff} & \mathcal{M}, g, i \not\models \phi \\
\mathcal{M}, g, i \models \phi_1 \wedge \phi_2 & \text{iff} & \mathcal{M}, g, i \models \phi_1 \text{ and } \mathcal{M}, g, i \models \phi_2 \\
\mathcal{M}, g, i \models \bigcirc\phi & \text{iff} & \mathcal{M}, g, i+1 \models \phi \\
\mathcal{M}, g, i \models \ominus\phi & \text{iff} & \mathcal{M}, g, i-1 \models \phi \\
\mathcal{M}, g, i \models \phi_1 \mathsf{U} \phi_2 & \text{iff} & \exists k \geq i : \mathcal{M}, g, k \models \phi_2 \text{ and} \\
 & & \qquad \forall j \text{ such that } i \leq j < k : \mathcal{M}, g, j \models \phi_1 \\
\mathcal{M}, g, i \models \phi_1 \mathsf{S} \phi_2 & \text{iff} & \exists k \leq i : \mathcal{M}, g, k \models \phi_2 \text{ and} \\
 & & \qquad \forall j \text{ such that } i \geq j > k : \mathcal{M}, g, j \models \phi_1 \\
\mathcal{M}, g, i \models x & \text{iff} & m_i = g(x) \\
\mathcal{M}, g, i \models n & \text{iff} & V(n) = \{m_i\} \\
\mathcal{M}, g, i \models @_s\phi & \text{iff} & \exists\, m_j \in [V, g](s) \text{ and } \mathcal{M}, g, j \models \phi \\
\mathcal{M}, g, i \models \downarrow x.\phi & \text{iff} & \mathcal{M}, g[x \mapsto m_i], i \models \phi \\
\mathcal{M}, g, i \models \exists_{p(x)}\, \phi & \text{iff} & \exists\, m_j \in V(p)(m_i) \text{ such that } \mathcal{M}, g[x \mapsto m_j], i \models \phi
\end{array}
$$

Figure 3: Infinite-path semantics of the logic

m_i is able to determine that the formula holds. For example, if p is true in m_2, then even though $\bigcirc p$ holds in m_1, an agent in m_1 would not normally be able to conclude this, since it cannot see into the future.

We deal with this by using a simplified form of the operator $\mathsf{Trunc}_\mathsf{S}$ from Eisner *et al.* [12]. A formula $\mathsf{Trunc}_\mathsf{S}\,\phi$ is true at a given state in a model if and only if ϕ can be shown to hold without any knowledge of future states. We define this formally as:

$$\mathcal{M}, g, i \models \mathsf{Trunc}_\mathsf{S}\,\phi \quad \text{iff} \quad \mathcal{M}^i, g, i \models^+ \phi$$

where $\models^+$ represents the *strong semantics* of Eisner *et al.* (defined below), and $\mathcal{M}^i$ is defined as follows. Let $\mathcal{M} = \langle\langle m_1 \ldots m_i \ldots\rangle, V\rangle$. For basic (nullary) propositions we define $V^i(p) = V(p) \setminus \{m_{i+1}, \ldots\}$, that is, V^i gives the same results as V, but without states m_j for $j > i$. As we assume that state-referencing propositions are only used to refer to the past, for propositions p of that type, we define $V^i(p) = V(p)$. Note that V^i includes all propositions and nominals in V, but loses any valuation information about them for the states m_j where $j > i$. We then define

Condition	$\mathcal{M}, g, i \models^{+} \phi$?	$\mathcal{M}, g, i \models^{-} \phi$?
$i > \lvert\mathcal{M}\rvert$	false	true
$i \leq \lvert\mathcal{M}\rvert$ and $\phi = \neg\psi$	iff $\mathcal{M}, g, i \not\models^{-} \psi$	iff $\mathcal{M}, g, i \not\models^{+} \psi$
$i \leq \lvert\mathcal{M}\rvert$ and $\phi = @_n\psi$	iff $\exists\, m_j : V(n) = \{m_j\}$ and $\mathcal{M}, g, j \models^{+} \psi$	iff $V(n) = \emptyset$ or $\exists\, m_j : V(n) = \{m_j\}$ and $\mathcal{M}, g, j \models^{-} \psi$
otherwise	As for $\models$, but substitute $\models^{+}$ or $\models^{-}$ (respectively) for $\models$ in recursive definitions	

Figure 4: Strong and weak semantics on finite paths

$\mathcal{M}^i = \langle\langle m_1 \ldots m_i\rangle, V^i\rangle$. We write $i > |\mathcal{M}|$ to test for states that have been pruned, i.e. if $i > |\mathcal{M}|$ then there is no m_i in $\mathcal{M}$. We write $i \leq |\mathcal{M}|$ to test for states that do exist, i.e. if $i \leq |\mathcal{M}|$ then $m_i \in M$ (where $\mathcal{M} = \langle M, V\rangle$).

We need to use the strong semantics ($\models^{+}$) because the standard semantics is defined over infinite sequences of states and does not provide any way to disregard information from future states. The strong semantics is skeptical: it concludes that $\mathcal{M}, g, i \models^{+} \phi$ only when there is enough evidence so far to definitely conclude that ϕ holds. To define negation, we also need its *weak* counterpart, $\models^{-}$. The weak semantics is generous: it concludes that $\mathcal{M}, g, i \models^{-} \phi$ whenever there is no evidence against ϕ so far.

Figure 4 defines the strong and weak semantics. Note that the semantics of negation switch between the strong and weak semantics: we can conclude *strongly* (respectively weakly) that $\neg\phi$ holds if and only if we can conclude *weakly* (respectively strongly) that ϕ does not hold. For example, suppose an agent is examining state $i-1$ of a trace in order to determine whether $\neg\bigcirc\phi$ holds. Let us assume that state $i-1$ is the final state of the trace that is available. Since the next state is not available ($i > |\mathcal{M}|$), we have that ϕ does not hold strongly in state i ($\mathcal{M}, g, i \not\models^{+} \phi$) as there is no definite evidence for ϕ; but that ϕ *does* hold weakly in state i ($\mathcal{M}, g, i \models^{-} \phi$), because there is no definite evidence *against* ϕ. We therefore have that $\mathcal{M}, g, i-1 \not\models^{+} \bigcirc\phi$ and that $\mathcal{M}, g, i-1 \models^{-} \bigcirc\phi$. Now, the agent has concluded (due to lack of evidence) that $\bigcirc\phi$ holds, but this is only a weak conclusion. What can it say about $\neg\bigcirc\phi$? Since the agent is reasoning based on the lack of information, any conclusions must be weak. Therefore, it should not conclude that $\mathcal{M}, g, i \models^{+} \neg\bigcirc\phi$ from $\mathcal{M}, g, i \not\models^{+} \bigcirc\phi$. However, it can reasonably conclude that $\mathcal{M}, g, i \models^{-} \neg\bigcirc\phi$. This behaviour is exactly what is given by the rules of Figure 4,

specifically:

$$\mathcal{M}, g, i \models^{+} \neg\phi \quad \text{iff} \quad \mathcal{M}, g, i \not\models^{-} \phi$$
$$\mathcal{M}, g, i \models^{-} \neg\phi \quad \text{iff} \quad \mathcal{M}, g, i \not\models^{+} \phi$$

The semantics for $@_n\psi$ in Figure 4 considers the case where the state referred to by the nominal n does not exist. In this case the weak semantics will consider $@_n\psi$ to hold, but the strong semantics will consider $@_n\psi$ to not hold.
We can now use the $\mathsf{Trunc}_{\mathsf{S}}$ operator to define fulfilment and violation:

$$\mathsf{Fulf}\,\phi \equiv \mathsf{Exp}\,\phi \wedge \mathsf{Trunc}_{\mathsf{S}}\,\phi$$
$$\mathsf{Viol}\,\phi \equiv \mathsf{Exp}\,\phi \wedge \mathsf{Trunc}_{\mathsf{S}}\,\neg\phi$$

This defines $\mathsf{Fulf}\,\phi$ to hold if an existing expectation of ϕ is true, and furthermore, if it can be known to be true without knowledge of future states. Similarly, $\mathsf{Viol}\,\phi$ is true if an existing expectation of ϕ becomes violated (i.e. $\neg\phi$), and no future states are needed to detect the violation.

4. Formula Progression

As outlined in Section 2, we use the notion of *formula progression* to describe how an unfulfilled and non-violated expectation evolves from one state to the next. Formula progression was introduced in the TLPlan [7] planner to allow "temporally extended goals" to be used to control the system's search for a plan. Rather than just describing the desired goal state for the plan to bring about, TLPlan used a linear temporal logic formula to constrain the path of states that could be followed while executing the plan. As planning proceeds, whenever a new action is appended to the end of the plan, the goal formula must be "progressed" to represent the residual constraint left once planning continues from the state resulting from executing that action.

Bacchus and Kabanza considered progression as a function mapping a formula and state to another formula, and defined this function inductively on the structure of formulae in their logic $\mathcal{LT}$, a first-order version of linear temporal logic. They proved the following theorem.

Theorem (Bacchus and Kabanza [7]) *Let $M = \langle w_0, w_1, \ldots\rangle$ be any $\mathcal{LT}$ model. Then, we have for any $\mathcal{LT}$ formula f in which all quantification is bounded, $\langle M, w_i\rangle \models f$ if and only if $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f, w_i)$.*

In other words, the truth of a linear temporal logic formula at a given point on a history of states is equivalent to the truth of the progressed formula at the next state in the history. This provides an incremental way of evaluating future-oriented temporal formulae.

In the theorem above, $\mathit{Progress}(f, w_i)$ is a meta-logical function. We wish to define progression as an operator within the logic, and so adapt the above theorem to provide a definition of a modal operator $\mathsf{Progress}(\phi, \psi)$:

$$\mathcal{M}, g, i \models \mathsf{Progress}(\phi, \psi) \text{ iff } \forall \mathcal{M}' \in \overline{\mathcal{M}}(i),\ \mathcal{M}', g, i \models \phi \iff \mathcal{M}', g, i{+}1 \models \psi$$

where $\overline{\mathcal{M}}(i)$ is the set of all possible infinite models that are extensions of $\mathcal{M}^i$ ($\mathcal{M}$ truncated at i) and which contain all the nominals in $\mathcal{M}$, including those at indices past i (although the states named by the nominals are not required to have a consistent index across all the extensions). The models of $\overline{\mathcal{M}}(i)$ need not agree with $\mathcal{M}$ on the truth of propositions for state indices $j > i$.

The intuition behind this definition is that, as shown in the definition of Bacchus and Kabanza, the amount of information given by a formula and the history up to the present state should be the same as that given by the progressed formula and the history extended by one step. No future information should be considered when progressing a formula, and this is modelled in our definition by considering all futures as possible, provided that the set of nominals in the models remains unchanged—this is necessary to allow the progression of $@_s\phi$ to be well defined.

We can then obtain the theorems of Figure 5, which define an inductive procedure for evaluating progression[6], in conjunction with the use of Boolean simplification to eliminate $\bot$ and $\top$ as subformulae. Note that the first two cases take precedence over the remaining rules: if a complex formula is known, based only on past information, to be true (or false), then it can be replaced with its valuation. In the figure, we denote subsitution of the nominal n for the free occurrences of x by $\phi[x/n]$. This evaluation procedure is similar to the function of Bacchus and Kabanza, but extended to account for the hybrid features of our logic. The theorems for the $\ominus$, S, $\downarrow$ and $\exists$ operators require there to be nominals naming the bound states; however, for our model checking application, this can be easily ensured by generating nominals of the form s_i when required. If a state has more than one nominal, any one can be chosen.

[6]The theorems for $\ominus$ and S, and the second case for $@_s\phi$, are the best that can be done compositionally, i.e. without knowledge of their subexpressions (which may be impossible to evaluate at the current state if they contain nested future-oriented expressions).

$\mathcal{M}, g, i \models \mathsf{Progress}(\phi, \top)$ if $\mathcal{M}, g, i \models \mathsf{Trunc}_{\mathsf{S}}\, \phi$

$\mathcal{M}, g, i \models \mathsf{Progress}(\phi, \bot)$ if $\mathcal{M}, g, i \models \mathsf{Trunc}_{\mathsf{S}}\, \neg\phi$

Otherwise:

$\mathcal{M}, g, i \models \mathsf{Progress}(\phi_1 \wedge \phi_2, \psi_1 \wedge \psi_2)$ if $\mathcal{M}, g, i \models \mathsf{Progress}(\phi_1, \psi_1)$ and $\mathcal{M}, g, i \models \mathsf{Progress}(\phi_2, \psi_2)$

$\mathcal{M}, g, i \models \mathsf{Progress}(\neg\phi, \neg\psi)$ if $\mathcal{M}, g, i \models \mathsf{Progress}(\phi, \psi)$

$\mathcal{M}, g, i \models \mathsf{Progress}(\bigcirc\phi, \phi)$

$\mathcal{M}, g, i \models \mathsf{Progress}(\phi_1 \,\mathsf{U}\, \phi_2, \psi_2 \vee (\psi_1 \wedge (\phi_1 \,\mathsf{U}\, \phi_2)))$ if $\mathcal{M}, g, i \models \mathsf{Progress}(\phi_1, \psi_1)$ and $\mathcal{M}, g, i \models \mathsf{Progress}(\phi_2, \psi_2)$

$\mathcal{M}, g, 1 \models \mathsf{Progress}(\ominus\phi, \bot)$

$\mathcal{M}, g, i \models \mathsf{Progress}(\ominus\phi, @_n\phi)$ if $i > 1$ where $V(n) = \{m_{i-1}\}$

$\mathcal{M}, g, i \models \mathsf{Progress}(\phi_1 \,\mathsf{S}\, \phi_2, @_n(\phi_1 \,\mathsf{S}\, \phi_2))$ where $V(n) = \{m_i\}$

$\mathcal{M}, g, i \models \mathsf{Progress}(@_s\phi, \psi)$ if $[V, g](s) = \{m_i\} \wedge \mathcal{M}, g, i \models \mathsf{Progress}(\phi, \psi)$

$\mathcal{M}, g, i \models \mathsf{Progress}(@_s\phi, @_s\phi)$ Used when the previous case does not apply

$\mathcal{M}, g, i \models \mathsf{Progress}(\downarrow x.\phi, \psi)$ if $\mathcal{M}, g, i \models \mathsf{Progress}(\phi[x/n], \psi)$ where $V(n) = \{m_i\}$

$\mathcal{M}, g, i \models \mathsf{Progress}(\exists_{p(x)}\, \phi, \psi)$ if $\Big(\forall j, m_j \in V(p)(m_i) :$ $\mathcal{M}, g, i \models \mathsf{Progress}(\phi[x/n_j], \psi_j)$ where $V(n_j) = \{m_j\}\Big)$ and $\psi \equiv \bigvee_{m_j \in V(p)(m_i)} \psi_j$

Figure 5: Recursive evaluation of the progression operator

5. Applying Model Checking to Expectation Monitoring

Model checking is the problem of determining for a *particular* model of a logical language whether a given formula holds in that model. Thus it differs from logical inference mechanisms which make deductions based on rules that are valid in *all* possible models. This makes model checking more tractable in general than deduction.

Model checking is commonly used for checking that models of dynamic systems, encoded as finite state machines, satisfy properties expressed in a temporal logic. However, model checking can also be applied to checking properties of *linear* models, e.g. sequences of observed states, and we have therefore investigated the applicability of model checking as a way of checking for expectations, fulfilments and violations over a model which is a linear history of observed states. This was done by extending an existing model checker, described in the next section.

5.1. The Hybrid Logics Model Checker

The Hybrid Logics Model Checker (HLMC) [2] implements the MCLITE and MCFULL labelling algorithms of Franceschet and de Rijke [8]. HLMC reads a model encoded in XML and a formula given in a textual notation, and uses the selected labelling algorithm to determine the label, true ($\top$) or false ($\bot$), for the input formula in each state of the model. It then reports to the user *all* the states in which the formula is true (i.e. it is a *global* model checker).

The two labelling algorithms are defined over a propositional temporal logic with the operators **F** ("some time in the future"), **P** ("some time in the past"), the binary temporal operators **U** (until) and **S** (since), the universal modality **A**, and the following features of hybrid logic: nominals, state variables, the operator $@_s$, and the binding operators $\downarrow x$ and $\exists x$ ("binding x to some state makes the following expression true"). The duals of the modal operators are defined in the usual way. The underlying accessibility relation is assumed to represent "later than" (the transitive closure of the "next state" relation underlying many temporal logics), so the definitions of **F** and **P** in terms of the accessibility relation are equivalent to those of our $\bigcirc$ and $\ominus$, respectively, over a "next state" accessibility relation. Time is not constrained to be linear.

The global model checking problem for any subset of this language that freely combines temporal operators with binders is known to be PSPACE-complete [8]. MCLITE is a bottom-up labelling algorithm for the sublanguage that excludes the two binding operators, and it runs in time $O(k.n.m)$ where k is the length of

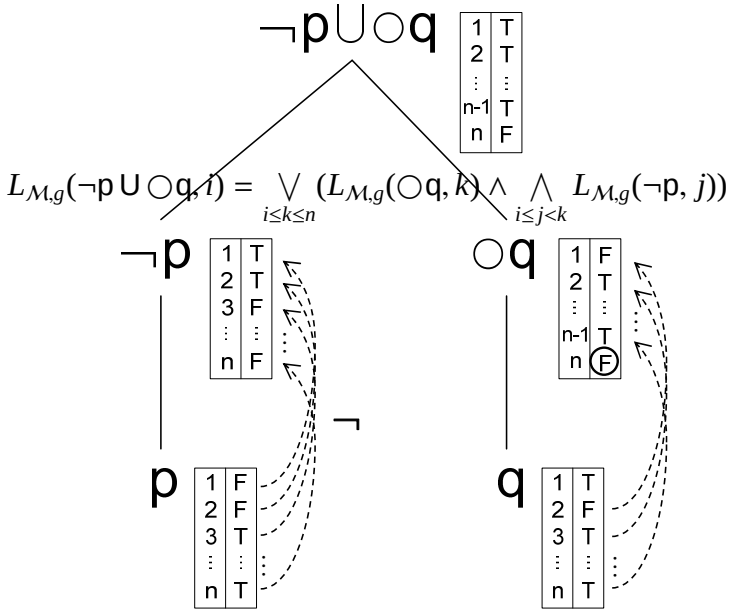


Figure 6: Example label computation in HLMC

the formula to be checked, n is the number of states in the model, and m is the size of the model's accessibility relation. MCFULL handles the full language, uses polynomial space, and runs in time exponential on the nesting degree of the binders in the formula. Specifically, let $r_{\downarrow}$ denote the nesting degree of $\downarrow$. Then the complexity is $O(k.n.m.n^{r_{\downarrow}})$. Since we are monitoring expectations in traces, we have models that are linear, i.e. each world is followed by at most one other world, and so $O(m) = O(n)$, and hence the complexity is $O(k.n^2)$ without the binding operator, and $O(k.n^{r_{\downarrow}+2})$ if the binding operator is used.

MCLITE works by labelling each subformula of the formula to be checked, for all states in the model, in a bottom-up manner. Figure 6 illustrates this process for an example formula. For each subformula, its Boolean value in each state (its *label*) is calculated, and a labelling function for the parent formula's outermost operator is then used to generate the label for that formula as a function of the subformulae's labels. For example, as shown in the figure, the labels for a formula $\neg\phi$ are produced by negating the labels for ϕ, the labels for $\bigcirc\phi$ are the labels for ϕ moved forward one state (with a new label of *false* generated for the final state), and the labels for $\phi \cup \psi$ are computed from the labels of ϕ and ψ using the Boolean expression shown.

Nominals and state variables

$$\mathcal{M}, g, i \models s \quad \text{iff} \quad [V, g](s) = \{m_i\}$$

$$L_{\mathcal{M},g}(s, i) \quad = \quad \begin{cases} \top & \text{if } [V, g](s) = \{m_i\} \\ \bot & \text{otherwise} \end{cases}$$

Operator $@_s$

$$\mathcal{M}, g, i \models @_s\phi \quad \text{iff} \quad \mathcal{M}, g, j \models \phi \text{ where } [V, g](s) = \{m_j\}$$

$$L_{\mathcal{M},g}(@_s\phi, i) \quad = \quad L_{\mathcal{M},g}(\phi, j) \text{ where } [V, g](s) = \{m_j\}$$

Operator $\bigcirc$

$$\mathcal{M}, g, i \models \bigcirc\phi \quad \text{iff} \quad i < |\mathcal{M}| \wedge \mathcal{M}, g, i+1 \models \phi$$

$$L_{\mathcal{M},g}(\bigcirc\phi, i) \quad = \quad (i < |\mathcal{M}| \wedge L_{\mathcal{M},g}(\phi, i+1))$$

Figure 7: The MCLITE labelling function (partial definition)

Figure 7 defines the semantics of some of the operators supported by HLMC together with the corresponding definition of the label, denoted $L_{\mathcal{M},g}(\phi, i)$. The presentation is adapted from that of Franceschet and de Rijke [8] to correspond to the HLMC operators, and to provide a declarative rather than procedural account[7]. It can be seen that in these cases the labelling function is a straightforward translation from the semantics in Figure 3—a property we have sought to preserve where possible for our extended notion of labels presented in Section 5.2.

The simple bottom-up labelling procedure does not work when binders are included in the language as there will be subformulae containing free state variables, and the values of these depend on the enclosing binding context. Instead, the recursive top-down MCFULL procedure is used. A formula is labelled by first labelling its immediate subformulae recursively, and then applying the appropriate labelling algorithm for the formula's operator. For operators in the MCLITE sublanguage, the MCLITE labelling algorithm is used. When the recursion encounters a formula of the form $\downarrow x.\phi_x$, the recursive global labelling is performed for *each* binding of x to a state in the model. Consider the formula $\mathbf{F}\downarrow x.\mathbf{G}(p \leftrightarrow @_x p)$ (eventually there is a state after which all states have the same value of p): labelling this for any given state s requires the truth of $\mathbf{G}(p \leftrightarrow @_x p)$ to be known

[7] For consistency with the rest of the paper we use the notation defined in Section 3 and assume that models are sequences of states. HLMC does not, in fact, restrict models to be sequences, but for our application to monitoring observed traces this is an appropriate restriction.

for an indeterminate number bindings of x to future states.

Franceschet and de Rijke claim that "MCFULL can be viewed as a general model checker for the hybridization of *any* temporal logic" by adding appropriate labelling subprocedures for each modal operator [8], and therefore the model checker HLMC was chosen as the basis for this research. However, HLMC is not a direct implementation of MCLITE and MCFULL. It is not specialised to temporal logic as it allows multiple accessibility relations to appear in the model and makes no assumptions about the structure of these relations. Formulae to be checked can include diamond and box modalities for each accessibility relation in both forward and reverse directions, the existential and universal modalities, and the hybrid logic operators $\downarrow x$ and $@_s$. There is (in the unmodified version) no support for **U** and **S**.

5.2. Handling Trunc_S

We have adapted HLMC for checking the fulfilment and violation of expectations. We assume (and verify) that the input model represents a linear path and thus contains a single "next state" accessibility relationship.

To allow the checking of fulfilment and violation, a labelling algorithm for Trunc_S was developed. This was complicated by the presence of past-time operators. Consider the label for $\mathsf{Trunc}_S \ominus\neg\phi$. Based on the definitions of Section 3, we have:

$$\begin{aligned} \mathcal{M},g,i \models \mathsf{Trunc}_S \ominus\neg\phi &\iff \mathcal{M}^i,g,i \models^+ \ominus\neg\phi \\ &\iff \mathcal{M}^i,g,i{-}1 \models^+ \neg\phi \\ &\iff \mathcal{M}^i,g,i{-}1 \not\models^- \phi \end{aligned}$$

or equivalently:

$$\begin{aligned} L_{\mathcal{M},g}(\mathsf{Trunc}_S \ominus\neg\phi, i) &= L^+_{\mathcal{M}^i,g}(\ominus\neg\phi, i) \\ &= L^+_{\mathcal{M}^i,g}(\neg\phi, i{-}1) \\ &= \neg L^-_{\mathcal{M}^i,g}(\phi, i{-}1) \end{aligned}$$

where $L^+_{M,g}$ and $L^-_{M,g}$ denote labelling under the strong and weak semantics, respectively. Thus to label $\mathsf{Trunc}_S \ominus\neg\phi$ at model index i it is necessary to know the weak semantics label for ϕ at index $i-1$ when the model is truncated at i. More generally, when labelling a formula ϕ at a model index i it is necessary to store both weak and strong labels with respect to all possible future truncation points:

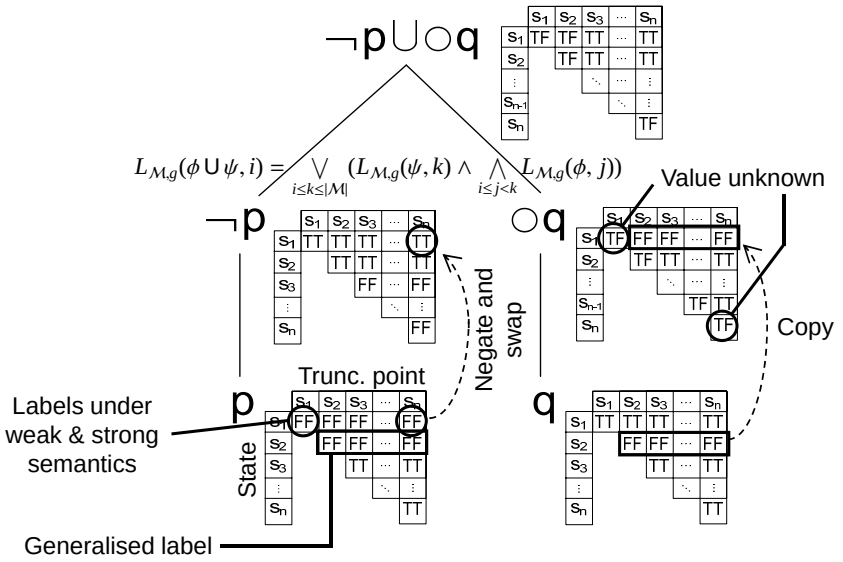


Figure 8: Example use of generalised labels

$L^-_{M^j,g}(\phi, i)$ and $L^+_{M^j,g}(\phi, i)$ for $j \geq i$. We therefore define a generalised label for a formula ϕ at model index i as a sequence of pairs of weak and strong labels for each possible truncation point from i to the final state in the model:

$$L_{\mathcal{M},g}(\phi, i) = \left\langle j : \left(L^-_{\mathcal{M}^j,g}(\phi, i),\ L^+_{\mathcal{M}^j,g}(\phi, i)\right) \mid i \leq j \leq |\mathcal{M}| \right\rangle$$

where $L^-_{\mathcal{M}^j,g}(\phi, i)$ is the value for ϕ at index i in the model under the weak semantics assuming a truncation at index j, and $L^+_{\mathcal{M}^j,g}(\phi, i)$ is the corresponding value under the strong semantics. Note that $L_{\mathcal{M},g}(\phi, i)$ is an empty sequence for $i > |\mathcal{M}|$.

For example, consider the formula $\phi = \bigcirc\bigcirc p$ and a model $\mathcal{M}$ with four states (s_1 to s_4) in which p holds only in s_2. The label for ϕ at state s_1 contains its truth values at s_1 under the weak and strong semantics for each of the four possible future truncation points (after s_1, s_2, s_3 and s_4). For the first two truncation points, the value of ϕ cannot be determined and the weak and strong semantics give different truth values. For truncation points s_3 and s_4, the formula is known to be false under both semantics. Therefore, we have (for any g) that $L_{\mathcal{M},g}(\phi, 1) = \langle 1 : (\top, \bot), 2 : (\top, \bot), 3 : (\bot, \bot), 4 : (\bot, \bot)\rangle$. The truth value $\bot$ for $\mathsf{Trunc}_\mathsf{S}\,\phi$ can then be read directly from the strong semantics component of the first element of this label.

Figure 8 illustrates the representation and calculation of generalised labels. The figure shows generalised labels as rows in diagonal matrices that are indexed

by the state at which evaluation is to be done and a possible (present or future) truncation point for the model. The functions to calculate each formula's labels from those of its suformulae are similar to those shown in Figure 6, but are extended to apply to generalised labels. However, we do not represent and compute with generalised labels in the explicit form shown in the figure. Instead we represent each generalised label $L_{\mathcal{M},g}(\phi, i)$ in an abbreviated form $L^a_{\mathcal{M},g}(\phi, i)$ that can omit weak/strong value pairs if there has been no change in these values since the previous truncation point in the sequence. Each entry in an abbreviated label is a truncation point and its associated weak and strong values:

$$L^a_{\mathcal{M},g}(\phi, i) = \langle j_1 : (w_{j_1}, s_{j_1}), \ldots, j_n : (w_{j_n}, s_{j_n}) \rangle$$

where $i = j_1 < \cdots < j_n \leq |\mathcal{M}|$ and $\forall_{1<i\leq n}\, j_i > j_{i-1} \wedge (w_{j_i}, s_{j_i}) \neq (w_{j_{i-1}}, s_{j_{i-1}})$. This abbreviated label defines the weak and strong labels for ϕ at each truncation point from i onwards as follows: $\forall_{1\leq k\leq n} \forall_{j_k \leq l < j_{k+1}} L^-_{\mathcal{M}^l,g}(\phi, i) = w_{j_k} \wedge L^+_{\mathcal{M}^l,g}(\phi, i) = s_{j_k}$ (with j_{n+1} defined as $|\mathcal{M}|+1$).

For example, the generalised label shown above for $\phi = \bigcirc\bigcirc p$ at state s_1 can be abbreviated to $\langle 1 : (\top, \bot), 3 : (\bot, \bot) \rangle$. In fact, we shall see later that abbreviated labels can be understood as a sequence of values in a three-valued logic (true, false and unknown yet) and always have at most two elements.

Conjunction and disjunction apply to generalised labels in a straightforward way, acting element-wise on unabbreviated labels (but the implementation for abbreviated labels must consider the omitted entries). These operations are defined for labels starting at the same index, i.e. $\langle i : (w_i, s_i), \ldots \rangle \wedge \langle j : (w'_i, s'_i), \ldots \rangle$ is syntactically correct if $i = j$. We write indexed conjunctions and disjunctions, e.g. $\wedge_{1\leq k\leq|\mathcal{M}|}^{\langle i:(w,s)\rangle}$, with the superscript label $\langle i : (w, s) \rangle$, indicating what the value is if there are no conjuncts or disjuncts.

Negation operates on the arguments and then exchanges the resulting weak and strong values at each truncation point, e.g. $\neg\, \langle 1 : (\top, \bot),\ 2 : (\top, \top) \rangle = \langle 1 : (\top, \bot), 2 : (\bot, \bot) \rangle$.

Abbreviated generalised labels are computed using a recursive procedure based on the labelling function defined in Figure 9. The second disjunct in the function for $\phi \,\mathsf{U}\, \psi$ deals with the case where ϕ is true in the current state and in all subsequent states. In this situation we must conclude that $\phi \,\mathsf{U}\, \psi$ is weakly true because there is no evidence against it. It may eventually become true if the model is extended with a new state in which ψ holds. The label $\langle i : (\top, \bot) \rangle$ is used as a mask to ensure that this disjunct only applies to the weak semantics.

Like the original version of HLMC, our extension is a *global* model checker—it computes labels for the target formula for *all* states. This is done by processing

Propositions

$$L^a_{\mathcal{M},g}(p,i) = \begin{cases} \langle i:(\top,\top)\rangle & \text{if } m_i \in V(p) \\ \langle i:(\bot,\bot)\rangle & \text{otherwise} \end{cases}$$

$$L^a_{\mathcal{M},g}(p(s),i) = \begin{cases} \langle i:(\top,\top)\rangle & \text{if } [V,g](s) \subseteq V(p)(m_i) \\ \langle i:(\bot,\bot)\rangle & \text{otherwise} \end{cases}$$

Hybrid logic constructs

$$L^a_{\mathcal{M},g}(s,i) = \begin{cases} \langle i:(\top,\top)\rangle & \text{if } [V,g](s) = \{m_i\} \\ \langle i:(\bot,\bot)\rangle & \text{otherwise} \end{cases}$$

$$L^a_{\mathcal{M},g}(@_s\phi,i) = \begin{cases} \pi_i(L^a_{\mathcal{M},g}(\phi,j)) & \text{if there exists } j \text{ such that } \{m_j\} = [V,g](s) \\ \langle i:(\top,\bot)\rangle & \text{otherwise} \end{cases}$$

where π_i 'projects' a label onto m_i:

$$\pi_i(l) = \begin{cases} \langle i:(\top,\bot)\rangle \bullet l & \text{if } l \text{ starts with a truncation point after } i \\ l \downharpoonright i & \text{otherwise,} \end{cases}$$

$\bullet$ is a label prepend operation that keeps the label in abbreviated form, $l \downharpoonright i = al(\langle j:(w_j,s_j) \in ul(l) \mid j \geq i\rangle)$, and *al* and *ul* abbreviate and unabbreviate labels (Note that $\downharpoonright$ can be *implemented* more efficiently).

$$L^a_{\mathcal{M},g}(\downarrow x.\phi,i) = L^a_{\mathcal{M},g[x\mapsto m_i]}(\phi,i)$$

$$L^a_{\mathcal{M},g}(\exists_{p(x)}\,\phi,i) = \bigvee^{i:\langle\bot,\bot\rangle}_{m_j \in V(p)(m_i)} L^a_{\mathcal{M},g[x\mapsto m_j]}(\phi,i)$$

Figure 9: Labelling temporal formulae (continued on next page)

Temporal operators

$$L^a_{\mathcal{M},g}(\bigcirc\phi, i) = \begin{cases} i:(\top,\bot) \bullet L^a_{\mathcal{M},g}(\phi, i{+}1) & \text{if } i < |\mathcal{M}| \\ \langle i:(\top,\bot)\rangle & \text{if } i = |\mathcal{M}| \end{cases}$$

$$L^a_{\mathcal{M},g}(\ominus\phi, i) = \begin{cases} L^a_{\mathcal{M},g}(\phi, i{-}1) \restriction i & \text{if } i > 1 \\ \langle i:(\bot,\bot)\rangle & \text{if } i = 1 \end{cases}$$

$$L^a_{\mathcal{M},g}(\phi \,\mathsf{U}\, \psi, i) = \bigvee^{i:\langle\bot,\bot\rangle}_{i\le k\le|\mathcal{M}|} \Big(\pi_i(L^a_{\mathcal{M},g}(\psi,k)) \wedge \bigwedge^{i:\langle\top,\top\rangle}_{i\le j<k} \pi_i(L^a_{\mathcal{M},g}(\phi,j))\Big)$$
$$\vee \Big(\langle i:(\top,\bot)\rangle \wedge \bigwedge^{i:\langle\top,\top\rangle}_{i\le j\le|\mathcal{M}|} \pi_i(L^a_{\mathcal{M},g}(\phi,j))\Big)$$

$$L^a_{\mathcal{M},g}(\phi \,\mathsf{S}\, \psi, i) = \Big(\bigvee^{i:\langle\bot,\bot\rangle}_{1\le k\le i} (\pi_i(L^a_{\mathcal{M},g}(\psi,k)) \wedge \bigwedge^{i:\langle\top,\top\rangle}_{k<j\le i} \pi_i(L^a_{\mathcal{M},g}(\phi,j)))\Big)$$

Figure 9: Labelling temporal formulae (continued)

the formula's syntax tree in a post-order fashion: each node is labelled for all states after labels for its subnodes have been computed, and is then stored as an attribute of the formula object. The exception is for the $\downarrow$ operator. When a formula $\downarrow x.\phi$ is encountered, ϕ is separately labelled under each possible binding of x to a state in the model, before the parent formula is labelled, as discussed in Section 5.1.

In fact, unless a specific application demands it, it is not necessary to perform *global* model checking using the labelling procedure. Figure 9 defines a recursive function that can be used to compute an abbreviated generalised label for a formula at a given state. Labelling a $\downarrow$ formula may still require multiple bindings to be tried, but the states considered for the binding will be governed by the labelling function definition for the parent operator (e.g. the definition for U controls which states j and k the subformulae should be labelled for). This approach is extended in Section 7.2 to provide an algorithm for online model checking.

In addition to our extension of HLMC, we have developed a new implementation in Python[8] (chosen to provide a succinct expression of the procedure, rather than efficient computation), and this supports both global model checking and labelling at a single state.

[8]This is available from the authors.

5.3. Computational complexity of label computation

In this section we analyse the size of abbreviated labels and the computational complexity of the labelling procedure.

Definition 1. *We define a label l for state i to be* simple *if it is in one of the following three forms:*

1. $\langle i : (x, x)\rangle$
2. $\langle i : (\top, \bot), j : (x, x)\rangle$
3. $\langle i : (\top, \bot)\rangle$

where x is either $\top$ or $\bot$ and $j > i$.

Intuitively, we can think of a generalised label for a formula ϕ in a given state s_i as representing two pieces of information: whether ϕ is true, false or unknown[9] in s_i, and, if it is true or false, in which state s_j ($j \geq i$) sufficient information will be available to determine this value. In other words, for each ϕ and s_i we have one of the following cases: (i) ϕ is known to be true or false in s_i (the first form above), (ii) based solely on information in s_i we cannot determine the truth of ϕ, but given future information in the states up to s_j ($j > i$) we can determine its truth (the second form), (iii) we cannot determine whether ϕ is true or false in s_i, and future information available in the full (finite) model does not help (the third form).

Note that labels (simple or not) never have pairs of the form $(\bot, \top)$, and in fact we can think of the pairs as encoding a three value logic[10] where (x, x) represents x (either $\top$ or $\bot$) and $(\top, \bot)$ represents the unknown value. The semantics for our logic on finite paths (Figure 4) imply that all labels must be simple: as later possible truncation points are considered when evaluating a formula, its value can change from unknown to true or false, but no other transitions are possible. This means that an entire simple label can be viewed as either "unknown" (no matter how many future states in the current truncated model are considered) or "true" or "false" (once a certain number of present and future states in the current truncated model are considered). For example, $\langle 1 : (\top, \bot)\rangle$ represents "unknown"

[9] Recall that we are considering finite models, so the values of future-oriented formulae may be unknown.

[10] Bauer *et al.* [13] independently observed that attempting to capture the outcome of checking a formula against a finite (possibly incomplete) trace is better modelled in terms of a three value logic that allows for an "unknown" value.

and $\langle 1 : (\top, \bot), 5 : (\bot, \bot)\rangle$ represents "false" (once the future truncation point 5 is available).

Although the semantics guarantee that all labels are simple, for our complexity results below we must show that our labelling function $L^a_{\mathcal{M},g}$ correctly produces only simple labels.

Theorem 1. *$L^a_{\mathcal{M},g}(\phi, i)$ produces simple labels.*
Proof: *See Appendix A*

We now consider the computational complexity of the labelling algorithm that has just been presented. Our algorithm differs from the original algorithm of Franceschet and de Rijke in that the structure of labels is more complicated in order to handle Trunc_S. However, rather than store up to n possible truncation points for each state, we use an abbreviated representation which (by Theorem 1) stores for each state either one or two state indices, each associated with two Boolean flags.

It is easy to see that computing the label for a single state (i.e. $L^a_{\mathcal{M},g}(\phi, i)$) for the base cases (state variables, nominals and propositions p and $p(s)$) is $O(1)$ and results in a label of length 1. Therefore computing the labels for all states in these cases is $O(n)$, for n states.

The complexity for most of the operators ($\neg$, $\vee$, $\wedge$, $@_s$, and $\ominus$) is $O(n.l)$ where l is the size of the resulting label (an exception is $\bigcirc$ which is $O(n)$). However, since we have shown that the abbreviated label size is actually constant, the complexity is just $O(n)$ for these connectives.

The complex cases are U and S (which are identical in terms of label size and complexity), as well as $\downarrow x$. Let us consider $\downarrow x$. It computes a complete labelling for the sub-formula n times, and hence has complexity $O(n.\mathcal{L})$ where $\mathcal{L}$ is the complexity of labelling the sub-formula. The complexity of $\exists_{p(x)}\,\phi$ follows similar lines: for each state we need to compute a complete labelling for the sub-formula some number of times. The actual number depends on which (previous) states make $p(x)$ hold. In the worse case (which is unlikely to occur with typical usage scenarios of state-referencing propositions) we may need to look at all previous states. Thus the complexity to label a single state is $O(n.\mathcal{L})$ and the complexity to label all states is $O(n^2.\mathcal{L})$.

The complexity of U and S is also a little more complicated. If we simply implement the definitions given in Figure 9 directly, we end up with a complexity of $O(l.n^3)$. To compute the label for $\phi\,\mathsf{U}\,\psi$ for a single state we need to consider all subsequent states (first conjunct), and for each of these consider all of the states

between it and the original state. This produces a Boolean formula with $O(n^2)$ terms. For each state considered we perform an adjustment operation (π) which has complexity $O(l)$. Overall, this gives $O(l.n^2)$ for a single state, and $O(l.n^3)$ for all states, which, since l is constant, is just $O(n^3)$.

This algorithm can be improved. For each state m_i to be labelled, as we scan forward to find a state m_k in which ψ holds, we can also maintain the conjunction of labels $L^a_{\mathcal{M},g}(\phi, j)$ over the states $\{m_j \mid 1 \leq j < k\}$. There is no need for a nested loop to compute this from scratch for each k. Also, many of the states being considered can be pruned. For example, consider labelling $p \mathsf{U} q$ in a model where q is only true once, at the very last state. Then we need to consider all subsequent states (first conjunct), but for all but one of these we don't need to consider any intermediate states since q is false. For the one case where q is true, we do need to consider all intermediate states. This gives $O(l.n^2)$ overall. If we now consider labelling $p \mathsf{U} q$ in a model where q is true in all states, then clearly we do not need to consider all subsequent states, since q is true in the current state.

Unfortunately, this is not true if we consider more complex formulae. If we consider $\phi \mathsf{U} \psi$, then in general we cannot stop at the first state in which ψ is true, but need to consider future all states. To see this, consider the formula $\omega = p \mathsf{U} (q \vee \bigcirc\bigcirc\bigcirc r)$ in a model where p is true for states s_1, s_2, s_3 and s_4, q is true in s_5, and r is true in s_6. We then have the following labels:

i	p	q	$\bigcirc\bigcirc\bigcirc r$	$q \vee \bigcirc\bigcirc\bigcirc r$
1	$\langle 1:(\top,\top)\rangle$	$\langle 1:(\bot,\bot)\rangle$	$\langle 1:(\top,\bot), 4:(\bot,\bot)\rangle$	$\langle 1:(\top,\bot), 4:(\bot,\bot)\rangle$
2	$\langle 2:(\top,\top)\rangle$	$\langle 2:(\bot,\bot)\rangle$	$\langle 2:(\top,\bot), 5:(\bot,\bot)\rangle$	$\langle 2:(\top,\bot), 5:(\bot,\bot)\rangle$
3	$\langle 3:(\top,\top)\rangle$	$\langle 3:(\bot,\bot)\rangle$	$\langle 3:(\top,\bot), 6:(\top,\top)\rangle$	$\langle 3:(\top,\bot), 6:(\top,\top)\rangle$
4	$\langle 4:(\top,\top)\rangle$	$\langle 4:(\bot,\bot)\rangle$	$\langle 4:(\top,\bot)\rangle$	$\langle 4:(\top,\bot)\rangle$
5	$\langle 5:(\bot,\bot)\rangle$	$\langle 5:(\top,\top)\rangle$	$\langle 5:(\top,\bot)\rangle$	$\langle 5:(\top,\top)\rangle$
6	$\langle 6:(\bot,\bot)\rangle$	$\langle 6:(\bot,\bot)\rangle$	$\langle 6:(\top,\bot)\rangle$	$\langle 6:(\top,\bot)\rangle$

Now, given these labels, we proceed to label ω for state s_1 by considering each state $s_1, s_2, \ldots$ in turn and checking whether $q \vee \bigcirc\bigcirc\bigcirc r$ holds. The first state in which this holds is s_3, which gives the label $\langle 1:(\top,\bot), 6:(\top,\top)\rangle$, computed as $\pi_1 L^a_{\mathcal{M},g}(q \vee \bigcirc\bigcirc\bigcirc r, 3) \wedge \pi_1 L^a_{\mathcal{M},g}(p, 1) \wedge \pi_1 L^a_{\mathcal{M},g}(p, 2)$. If we were to stop here, that would be the final label. However, if we continue to consider states s_4 and s_5 we find that $q \vee \bigcirc\bigcirc\bigcirc r$ is also true in state s_5, contributing to the label for ω the disjunct $\langle 1:(\top,\bot), 5:(\top,\top)\rangle$, computed as $\pi_1 L^a_{\mathcal{M},g}(q \vee \bigcirc\bigcirc\bigcirc r, 5) \wedge \pi_1 L^a_{\mathcal{M},g}(p, 1) \wedge \pi_1 L^a_{\mathcal{M},g}(p, 2) \wedge \pi_1 L^a_{\mathcal{M},g}(p, 3) \wedge \pi_1 L^a_{\mathcal{M},g}(p, 4)$. This new disjunct has an earlier truncation point at which the value of ω is known.

This shows that when labelling $\phi \mathsf{U} \psi$, if we stop when we find a state in which ψ is true, we risk missing out on a more precise label. However, even though we cannot stop at the first state in which ψ is true, the resulting complexity for labelling $\phi \mathsf{U} \psi$ for all states is not affected. The complexity of the optimised process for labelling $\phi \mathsf{U} \psi$ is $O(n^2)$.

Thus the overall computational complexity is as follows, where k is the size of the formula being checked, and n is the number of states in the model.

- $O(k.n^2)$ if $\downarrow x$ and $\exists_{p(x)}$ are not used.
- $O(k.n^{r_\downarrow+(2r_\exists)+2})$ if $\downarrow x$ is used $r_\downarrow$ times and $\exists_{p(x)}$ is used $r_\exists$ times.

At this point it is worth considering whether these complexity results could be improved by applying a *symbolic* model checking technique. Symbolic model checking is a technique that was developed to avoid the state explosion problem in checking models that are defined by a finite (but possibly very large) state machine. Instead of explicitly representing the transitions of the finite state machine, symbolic model checkers encode the transition system as a propositional logic formula, and the use of disjunction in this formula allows multiple transitions to be encoded in a compact and implicit way. However, the problem we are addressing is different from that of checking a finite state machine. Our models are sequences of observed states, and we know nothing about the process that generated these states (they could, in fact, be generated by human activity). Even if we observe multiple states (or even sequences of states) that appear identical, we cannot collapse these states or sequences into one. Therefore, there is no common structure that could be exploited by a symbolic representation of our models, and the technique seems unlikely to be beneficial.

5.4. Defining Expectation, Fulfilment and Violation

We now show how the semantics of expectation, fulfilment and violation are defined and implemented in our model checker. We elaborate on the intuitive account of these notions given in Section 2. We wish to use the model checker to check for the existence of rule-based conditional expectations, and their fulfilments and violations without requiring the rules of expectation to be hard-coded in the model checker, or integrated into the labelling procedure dynamically. Therefore, we define a *hypothetical* expectation modality $\mathsf{Exp}(\lambda, \rho, n, \phi)$. This means (informally) that if there *were* a rule $\lambda \rightarrow \mathsf{Exp}(\rho)$ then λ would have been strongly true at a previous state named by nominal n, the rule would have fired, and the expectation ρ would have progressed (possibly over multiple intermediate states)

to ϕ in the current state. This means that we do not have to hardcode rules into the model checker, or provide a mechanism to read and internalise them. Instead, a rule of interest to the user can be supplied as arguments to an input formula using an ExistsExp, ExistsFulf or ExistsViol modality (defined below). First we define $\mathsf{Exp}(\lambda, \rho, n, \phi)$:

$$\begin{aligned}\mathcal{M}, g, i \models^{\pm} \mathsf{Exp}(\lambda, \rho, n, \psi) \text{ iff } & \mathcal{M}, g, i \models^{\pm} \mathsf{Trunc_S}\, \lambda,\ V(n) = \{m_i\} \text{ and } \psi = \rho \\ \text{or } \exists \phi \text{ s.t. } & \mathcal{M}, g, i-1 \models^{\pm} \mathsf{Exp}(\lambda, \rho, n, \phi), \\ & \mathcal{M}, g, i-1 \not\models^{\pm} \mathsf{Trunc_S}\, \phi, \\ & \mathcal{M}, g, i-1 \not\models^{\pm} \mathsf{Trunc_S}\, \neg\phi \text{ and} \\ & \mathcal{M}, g, i-1 \models^{\pm} \mathsf{Progress}(\phi, \psi)\end{aligned}$$

where we write $\models^{\pm}$ to indicate that the choice between the weak or strong semantics is immaterial as future states play no role in this definition.

The first disjunct in the definition expresses the case in which the hypothetical rule matches the current state. Note that we use $\mathsf{Trunc_S}$ when evaluating the rule's condition λ to restrict it to present and past information only. The second disjunct expresses the case of progressing a non-fulfilled and non-violated exectation from the previous state. In order to use nominals to name the state at which rules apply, we require that the model has nominals for each state. These can be generated implicitly by the model checker, e.g. s_1, s_2, ...

We also define hypothetical versions of Fulf and Viol as follows:

$$\mathcal{M}, g, i \models^{\pm} \mathsf{Fulf}(\lambda, \rho, n, \phi) \quad \text{iff} \quad \mathcal{M}, g, i \models^{\pm} \mathsf{Exp}(\lambda, \rho, n, \phi) \text{ and } \mathcal{M}, g, i \models^{\pm} \mathsf{Trunc_S}\, \phi$$

$$\mathcal{M}, g, i \models^{\pm} \mathsf{Viol}(\lambda, \rho, n, \phi) \quad \text{iff} \quad \mathcal{M}, g, i \models^{\pm} \mathsf{Exp}(\lambda, \rho, n, \phi) \text{ and } \mathcal{M}, g, i \models^{\pm} \mathsf{Trunc_S}\, \neg\phi$$

These modalities are not used directly by the model checker. Instead we define the following existential version of Exp:

$$\mathcal{M}, g, i \models^{\pm} \mathsf{ExistsExp}(\lambda, \rho) \text{ iff } \exists n, \phi \text{ s.t. } \mathcal{M}, g, i \models^{\pm} \mathsf{Exp}(\lambda, \rho, n, \phi)$$

with similar definitions for $\mathsf{ExistsFulf}(\lambda, \rho)$ and $\mathsf{ExistsViol}(\lambda, \rho)$. These correspond to the actual queries that we wish to make to the model checker: "Are there any expectations (or fulfilments or violations) resulting from the given rule?"

To compute labels for these existential modalities, we first compute the following *witness function* $W_{\mathcal{M},g,i}$ iteratively for i increasing from 1 to $|\mathcal{M}|$ (where labels for the subformulae λ and ρ have already been computed due to the top-

down recursive labelling algorithm, and progressed formulae are labelled as they are generated[11]):

$$W_{\mathcal{M},g,i}(\mathsf{ExistsExp}(\lambda,\rho)) = \left\{ \begin{array}{ll} \{(n,\rho)\} & \text{where } V(n)=\{m_i\} \\ & \text{if } \mathcal{M},g,i \models^{\pm} \mathsf{Trunc_S}\,\lambda \\ \emptyset & \text{otherwise} \end{array} \right\} \cup$$
$$\{(n,\psi) \mid \exists\,\phi.(n,\phi) \in W_{\mathcal{M},g,i-1}(\mathsf{ExistsExp}(\lambda,\rho)),$$
$$\mathcal{M},g,i-1 \not\models^{\pm} \mathsf{Trunc_S}\,\phi,$$
$$\mathcal{M},g,i-1 \not\models^{\pm} \mathsf{Trunc_S}\,\neg\phi \text{ and}$$
$$\mathcal{M},g,i-1 \models^{\pm} \mathsf{Progress}(\phi,\psi)\}$$

This collects all pairs (n,ϕ) making $\mathsf{Exp}(\lambda,\rho,n,\phi)$ true at i for a given λ and ρ. The corresponding label for $\mathsf{ExistsExp}(\lambda,\rho)$ at i is then $\langle i:(\bot,\bot)\rangle$ if the witness set is empty, and $\langle i:(\top,\top)\rangle$ otherwise. Note that the (abbreviated) generalised labels discussed in Section 5.2 can be used directly to evaluate the $\mathsf{Trunc_S}$ formulae. Example witnesses for an ExistsExp formula are shown in Figure 10 (where we show labels in their unabbreviated form, for convenience).

Although we have defined Progress as a temporal operator, we do not support its use in the input language to the model checker. It is built in to the model checker as a formula-transforming function $progress(\mathcal{M},g,i,\phi)$, and we apply some Boolean simplications after progression to partially evaluate the result where $\top$ or $\bot$ appear as subformulae, nested negations appear, etc.

Witness functions are also defined for $\mathsf{ExistsFulf}(\lambda,\rho)$ and $\mathsf{ExistsViol}(\lambda,\rho)$ by taking the subset of pairs (n,ϕ) in $\mathsf{ExistsExp}(\lambda,\rho)$ for which $\mathsf{Trunc_S}\,\phi$ strongly holds and $\mathsf{Trunc_S}\,\neg\phi$ strongly holds, respectively.

Finally, we can use the model checker to check for expectations, violations and fulfilments over a given model by performing the global model checking procedure with an empty intial binding g for an input formula such as $\mathsf{ExistsExp}(\lambda,\rho)$, $\mathsf{ExistsFulf}(\lambda,\rho)$ or $\mathsf{ExistsViol}(\lambda,\rho)$, where condition λ and expectation ρ correspond to some rule of interest. The model checker will report all witnesses for the input formula for all states. This can be easily generalised to apply to disjunctions of input formulae referring to multiple rules. Note that although the witnesses for the ExistsExp modality could be used to generate labels for Exp for a given rule, we do not currently support the use of Exp to appear within rules, nor the use of nested ExistsExp formulae (although there should be no difficulties in allowing

[11] Progressed formulae may contain subformulae that are already labelled, and recursive labelling can stop at this point.

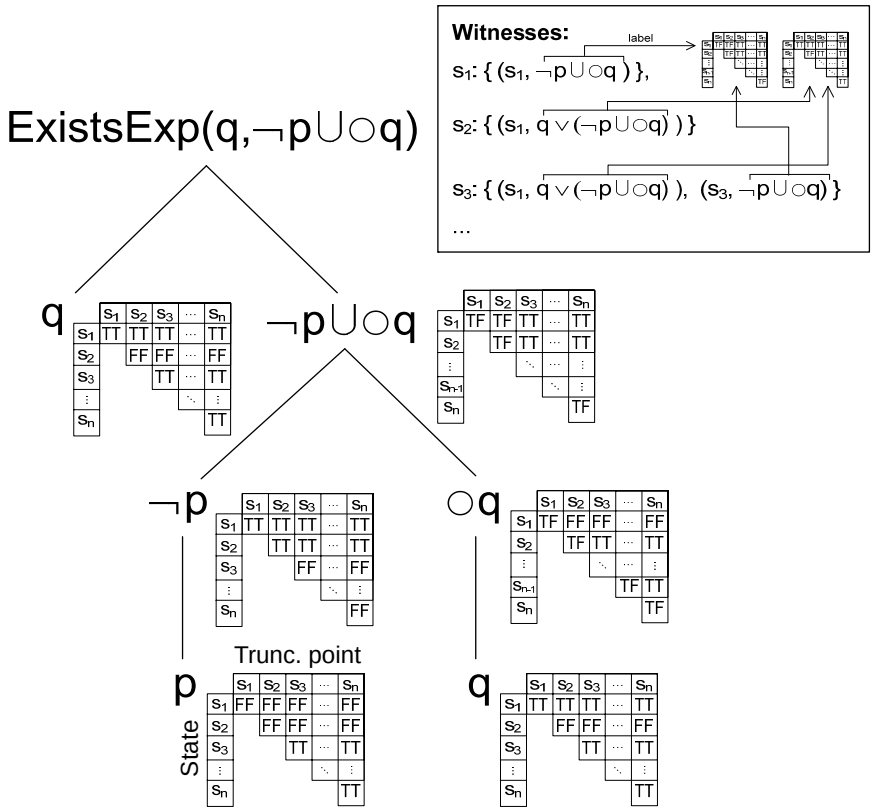


Figure 10: Example witnesses for ExistsExp

the latter to appear within the conditions of rules). Further investigation in this area is needed to allow the checking of interdependent expectations.

We now briefly consider the computational complexity of ExistsExp. In order to compute the witness set we scan forwards across each of the n states, which is $O(n)$. Checking whether $\mathsf{Trunc}_S\,\lambda$ holds is assumed to be constant (because labelling has already been done[12]). In each state we need to progress each of the active expectations. In the worse case there can be n expectations (since at most one expectation can be triggered in each state), and so we have $O(p.n^2)$ where p is the cost of progressing a formula. Since progression is compositional, we have

[12]In fact, for progressed formulae, some labelling may need to be done, but even if we assume that a complete labelling needs to be done, it still does not affect the overall complexity of ExistsExp.

that p is bounded by the size of the formulae, k, giving overall $O(k.n^2)$.

6. Examples

In this section we consider two examples: a hypothetical travel agent scenario and a real application of the model checker to monitoring actions in the Second Life virtual world.

6.1. The Acme travel agent scenario

Consider the Acme travel agent scenario, which we quote from Snell [14]:

1. Acme Travel receives an itinerary from Karla, the customer.
2. After checking the itinerary for errors, the process determines which reservations to make, sending simultaneous requests to the appropriate airline, hotel and car rental agencies to make the appropriate reservations.
3. If any of the three reservation tasks fails, the itinerary is cancelled by performing the "compensate" activity and Karla is notified of the problem.
4. Acme Travel waits for confirmation of the three reservation requests.
5. Upon receipt of confirmation, Acme Travel notifies Karla of the successful completion of the process and sends her the reservation confirmation numbers and the final itinerary details.
6. Once Karla is notified of either the success or failure of her requested itinerary, she may submit another travel request.

In this section we consider two possible conditional constraints that might be desirable to check in this scenario. For a range of other possible constraints, see van der Aalst and Pesic [15] who have formalised the scenario as a workflow (using temporal logic).

Perhaps most important for Karla is that billing should only ever take place if all three reservation tasks succeed, which can be formalised as a rule saying that if her card is charged (*ccard*) then we expect that all three activities (airline, hotel and car) have already succeeded. We formalise the expectation that the three activities have succeeded as $\diamondminus airline\_ok \wedge \diamondminus hotel\_ok \wedge \diamondminus car\_ok$, which requires all three to have occurred in the past, but not necessarily at the same time.

Let us now consider a trace in which Acme mistakenly charges Karla's card before the car reservation is performed, and see how the model checker performs its checking. Our model has the following states:

1. *request*

2. *airline_ok*
3. *hotel_ok*
4. *ccard*

The formula that we check is:

$$\mathsf{ExistsViol}(\mathit{ccard}, \diamondsuit\!\!\!\!-\mathit{airline\_ok} \wedge \diamondsuit\!\!\!\!-\mathit{hotel\_ok} \wedge \diamondsuit\!\!\!\!-\mathit{car\_ok})$$

In order to check this formula, we compute witnesses (as described in the previous section) for $\mathsf{ExistsExp}(\mathit{ccard}, \rho)$, where $\rho = \diamondsuit\!\!\!\!-\mathit{airline\_ok} \wedge \diamondsuit\!\!\!\!-\mathit{hotel\_ok} \wedge \diamondsuit\!\!\!\!-\mathit{car\_ok}$. The only state in which *ccard* is true is state 4, and so we have in that state that $\mathcal{M}, g, 4 \models^{\pm} \mathsf{Trunc_S}\, \mathit{ccard}$ and hence that $W_{\mathcal{M},g,4}(\mathsf{ExistsExp}(\mathit{ccard}, \rho)) = \{(4, \rho)\}$. However, we also have that $\mathsf{ExistsViol}(\mathit{ccard}, \rho)$ holds at state 4 with witness set $\{(4, \rho)\}$ because we know that the expectation ρ is created there, and we have that $\mathcal{M}, g, 4 \models^{\pm} \mathsf{Trunc_S}\, \neg\rho$. To see this simply observe that:

$$\begin{array}{ll} & \mathcal{M}, g, 4 \models^{\pm} \mathsf{Trunc_S}\, \neg\rho \\ \iff & \mathcal{M}^4, g, 4 \models^{+} \neg\rho \\ \iff & \mathcal{M}^4, g, 4 \not\models \rho \\ \iff & \mathcal{M}^4, g, 4 \not\models \diamondsuit\!\!\!\!-\mathit{airline\_ok} \text{ or } \mathcal{M}^4, g, 4 \not\models \diamondsuit\!\!\!\!-\mathit{hotel\_ok} \text{ or } \mathcal{M}^4, g, 4 \not\models \diamondsuit\!\!\!\!-\mathit{car\_ok} \end{array}$$

Since none of the states in the model have *car_ok*, then $\mathcal{M}^4, g, 4 \not\models \diamondsuit\!\!\!\!-\mathit{car\_ok}$ holds. Hence at state 4 the checker concludes that the expectation is violated.

Let us now consider another possible expectation: that if any of the booking activities fails, then compensation is performed, and Karla is subsequently notified. This can be formalised as a rule saying that if $\mathit{airline\_fail} \vee \mathit{hotel\_fail} \vee \mathit{car\_fail}$ then we expect that $\Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified})$, i.e. that eventually compensation and notification are done, but that notification must be done after (or at the same time as) compensation. We consider a model with the following states:

1. *request*
2. *airline_ok*
3. *hotel_fail*
4. *compensate*
5. *notified*

The formula that we check is:

$$\mathsf{ExistsFulf}(\mathit{airline\_fail} \vee \mathit{hotel\_fail} \vee \mathit{car\_fail}\ ,\ \Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified}))$$

As before, we compute a witness set for $\mathsf{ExistsExp}(\cdots)$ for each state, and find that the rule is only fired in state 3, resulting in the expectation $\Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified})$ in that state.

In this case, the expectation is neither fulfilled nor violated in state 3, and so it is progressed using the following rule[13] (which can be derived from the rule for U, recalling that $\Diamond\phi \equiv \top \mathsf{U}\, \phi$):

$$\mathcal{M}, g, i \models \mathsf{Progress}(\Diamond\phi, \psi \vee \Diamond\phi) \text{ iff } \mathcal{M}, g, i \models \mathsf{Progress}(\phi, \psi)$$

When applied to $\Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified})$ in state 3 we have that

$$\mathcal{M}, g, 3 \models \mathsf{Progress}(\Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified}), \psi \vee \Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified}))$$

where ψ is the progression of $\mathit{compensate} \wedge \Diamond \mathit{notified}$, that is

$$\mathcal{M}, g, 3 \models \mathsf{Progress}(\mathit{compensate} \wedge \Diamond \mathit{notified}, \psi)$$

and since *compensate* is false in state 3, it progresses to $\bot$ and hence ψ is $\bot$ and the expectation as a whole is progressed to itself.

Since the expectation is neither fulfilled nor violated in state 4, it is progressed again:

$$\mathcal{M}, g, 4 \models \mathsf{Progress}(\Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified}), \psi \vee \Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified}))$$

where, as before, ψ is the progression of $\mathit{compensate} \wedge \Diamond \mathit{notified}$. However, in state 4 *compensate* is true, *compensate* is progressed to $\top$, and so ψ is just the result of progressing $\Diamond \mathit{notified}$. Using the rule for progressing $\Diamond$ we have that ψ is $\chi \vee \Diamond \mathit{notified}$ where χ is the result of progressing *notified*. Since *notified* is false in state 4, it progresses to $\bot$ and hence $\psi = \bot \vee \Diamond \mathit{notified} = \Diamond \mathit{notified}$, and the expectation as a whole is progressed to $\Diamond \mathit{notified} \vee \Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified})$.

We finally come to state 5. In this state we have that the expectation, which has been progressed to $\Diamond \mathit{notified} \vee \Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified})$, is fulfilled, since *notified* is true in state 5.

These witnesses are summarised in the table below.

[13] We use this derived rule for brevity. The same result is obtained by first expanding $\Diamond$ in terms of U.

State	Propositions	ExistsExp Witnesses
1	*request*	none
2	*airline_ok*	none
3	*hotel_fail*	$(3, \Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified}))$
4	*compensate*	$(3, \Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified}))$
5	*notified*	$(3, \Diamond \mathit{notified} \vee \Diamond(\mathit{compensate} \wedge \Diamond \mathit{notified}))$

None of the formulae in these witness sets are violated (i.e. found to be strongly false) in the associated state, and, in fact, in state 5 the associated witness formulae is strongly true, i.e. fulfilled.

6.2. Monitoring a soccer player in Second Life

A key precondition to applying the techniques presented in this paper, and in other approaches to monitoring social interactions, is to find a way of observing events of interest and mapping them to an appropriate symbolic representation. Satisfying this requirement is considerably simplified when the interactions to be observed are already conducted via an electronic medium that provides programmatic access to a symbolic description of events (at least to some degree). For this reason, we have been experimenting with our approach to modelling and monitoring social expectations in the context of the Second Life virtual world [16].

Second Life is an online virtual world hosted on a grid of servers. Users connect via a client application to control the movements of an in-world character (their "avatar") and view a representation of the simulated world in the neighbourhoood of the avatar. Users can communicate with others via text chat and voice, can rent virtual regions of land from the operators of Second Life (Linden Labs), and can create three-dimensional objects and textures within the virtual world. Objects can be animated by scripts written in an the event-driven Linden Scripting Language (LSL). This language also provides the ability to detect the presence and certain properties of other objects and avatars. Not only does this provide a good testbed for our work, but we also believe that there is great benefit to be gained by applying socially inspired tools from multi-agent systems to help Second Life users gain a greater awareness of the social context of the (virtual) actions performed in-world by them and others [17].

While our initial work on monitoring social expectations in Second Life focused on simple scenarios involving changes to avatars' "basic animations" (such as walking and standing), we are now investigating the more challenging task of monitoring expectations of avatars playing association football (soccer) in Second Life via the Second Life Football System [18]. This system provides scripted

stadium and ball objects that can be deployed inside Second Life, as well as a "head-up display" object that an avatar can wear to allow the user to initiate kick and tackle actions. Using a combination of techniques (an LSL script, a client built using the LibOpenMetaverse library [19], and the Esper complex event processing engine [20]), it is possible to obtain reliable real-time information on the positions of the ball and the players, determine which player (if any) is in possession of the ball, and detect kick and tackle events as well as complex derived events such as a pass being completed and a goal being scored[14]. This information is encoded as a sequence of states and sent to the model checker along with a property to be monitored.

Our aim is to model and monitor football rules and team strategies that involve multiple players. However, for the present we are considering simpler scenarios involving a single player. Consider, for example, the following training exercise that a coach might expect a player to complete:

> Starting in a certain region of the field ("zone 1"), the player must dribble the ball while advancing continuously down the field until another region ("zone 2") is reached. The player must then attempt to kick the ball into the goal from within zone 2. The exercise finishes if a goal is scored from this kick.

The model checker can monitor the fulfilment of this exercise for the coach using the following formula:

$$\begin{aligned}\mathsf{ExistsFulf}(&\neg \mathit{ex1\_achieved} \wedge \mathit{in\_zone1} \wedge \mathit{dribbling\_downfield} \\ &\qquad \wedge \neg\ominus(\mathit{in\_zone1} \wedge \mathit{dribbling\_downfield}), \\ &\mathit{dribbling\_downfield}\ \mathsf{U}\ (\mathit{in\_zone2} \wedge \mathit{kick} \wedge \downarrow x.\Diamond(\exists_{\mathit{goal}(y)}\, @_x\, y))\end{aligned}$$

The predicates used to encode this domain are as follows: *ex1_achieved* is a predicate asserting that this "Exercise 1" has been completed, and it is added to the model by the application connecting Second Life and the model checker once the rule has been activated and then fulfilled; *in_zone1* and *in_zone2* mean that the ball is in zone 1 or 2 (respectively); *dribbling_downfield* means that the player is in possession of the ball, and is moving towards the goal line; *kick* indicates that the player has just kicked the ball; and $\mathit{goal}(s)$ means that a goal has just been scored and this goal-scoring action was initiated by a kick in the state s. The expression

[14] The details of this process are beyond the scope of this paper.

$@_x\, y$ means that the (bound) variables x and y refer to the same state (recall that in hybrid logic, the formula y is true only in the state that y is bound to).

We treat scoring a goal as an event that occurs over an interval of time, rather than in a single state. It begins with the single-state event *kick*, continues with movement of the ball while no one is in possession, and is terminated by the ball entering the goal. Note that while there will be many kicks in a game of soccer, only a few of these result in goals, and only the last kick that occurred before the ball enters the goal should be treated as the start of a goal-scoring action. Therefore, the occurrence of $goal(s)$ can only be detected and recorded in the state at the end of the interval over which it took place.

The second line of the formula above is an optimisation—it prevents the expectation becoming active when an equivalent previous instance is already known to have been activated.

Figure 11 shows a scenario in which the player's performance of the training exercise is monitored, and a successful conclusion is reached. The predicates mentioned above are abbreviated as (respectively) *ea*, *iz1*, *iz2*, *dd*, *k* and $g(s)$. The nominal for state s_{56} is written as s56. In this scenario, the player begins outside zone 1, then enters the zone, starts dribbing the ball down the field, leaves zone 1, enters zone 2, and then kicks the ball. The ball enters the goal some time later. The rule defined by the arguments to ExistsExp is triggered in state s_{29}, and the resulting expectation is progressed from one state to the next until the resulting expectation is finally fulfilled in state s_{67}. Finally, the predicate *ea* is asserted by the application in state s_{68} to indicate the successful application of the rule[15].

7. Incremental Checking

Earlier we said that the labelling method worked equally well for online and offline model checking. This section discusses in more detail how online checking works. For the algorithms in this section we assume that whenever a label $l = L^q_{\mathcal{M},\emptyset}(\phi, i)$ is computed (where $\emptyset$ is the initial 'empty' binding function), the value is cached, e.g. as an entry $i \mapsto l$ in a labels map associated with ϕ. If ϕ is an ExistsExp, ExistsFulf or ExistsViol formula, the witnesses are also cached.

In contrast to offline monitoring, we do not seek to pre-compute labels at all states for all immediate subformulae of ϕ before labelling ϕ itself. Instead we calculate and cache labels only as needed.

[15] Details of the pragmatics of rule definition and management are outside the scope of this paper.

States	Preds. holding	Expectation	Fulfilled
s_1–s_{23}			
s_{24}–s_{28}	*iz1*		
s_{29}–s_{33}	*iz1*, *dd*	$dd \cup (iz2 \wedge k \wedge \downarrow x.\Diamond(\exists_{g(y)}@_x y))$	✗
s_{34}–s_{48}	*dd*	$dd \cup (iz2 \wedge k \wedge \downarrow x.\Diamond(\exists_{g(y)}@_x y))$	✗
s_{49}–s_{55}	*dd*, *iz2*	$dd \cup (iz2 \wedge k \wedge \downarrow x.\Diamond(\exists_{g(y)}@_x y))$	✗
s_{56}	*dd*, *iz2*, *k*	$dd \cup (iz2 \wedge k \wedge \downarrow x.\Diamond(\exists_{g(y)}@_x y))$	✗
s_{57}	*iz2*	$\Diamond(\exists_{g(y)}@_{s56} y) \vee$	
		$(dd \cup (iz2 \wedge k \wedge \downarrow x.\Diamond(\exists_{g(y)}@_x y)))$	✗
s_{58}–s_{59}	*iz2*	$\Diamond(\exists_{g(y)}@_{s56} y)$	✗
s_{60}–s_{66}		$\Diamond(\exists_{g(y)}@_{s56} y)$	✗
s_{67}	$g(s_{56})$	$\Diamond(\exists_{g(y)}@_{s56} y)$	✔
s_{68}	*ea*		

Figure 11: A successful conclusion of the Second Life football training exercise

Given a model $\mathcal{M}$ with states $m_1, \ldots, m_i$, a formula ϕ where $L^a_{\mathcal{M},\emptyset}(\phi, i)$ has been cached, a new state m_{i+1} to be appended to $\mathcal{M}$, and valuations for the propositions and nominals in m_{i+1}, the *single state* online model checking problem for $\mathcal{M}$ and ϕ is the problem of computing $L^a_{\mathcal{M},\emptyset}(\phi, i+1)$ using the cached labels for ϕ and its subformulae. This problem assumes that the application invoking the model checker only needs "real-time" information about whether ϕ holds for each state as it is added, and it is not required to recheck ϕ at any earlier state m_j. The labels for ϕ at these earlier states could, in fact, become out of date as the model is extended with new states. Consider the formula $\Diamond p$ in a finite model in which p never holds. This has the "unknown" label $\langle j : (\top, \bot)\rangle$ in every state m_j, but once a new state m_{i+1} in which p holds is appended to the model, these labels should all be changed to $\langle j : (\top, \bot),\ i+1 : (\top, \top)\rangle$. However, the application may not need to know about these changes if it is focused on the present.

We also define the *prefix update* online model checking problem. In this version of the problem we assume that before adding the new state, the labels for ϕ are up to date for *all* earlier states, as well as any subformulae labels that were needed during that computation. The problem is then to compute the label for ϕ at the new state m_{i+1} *and* to ensure that the labels for ϕ at all earlier states are updated to reflect the addition of state m_{i+1}.

There is a simple algorithm for the *single state* online model checking problem. As the new label $L^a_{\mathcal{M},\emptyset}(\phi, i+1)$ is computed, the recursive definitions of Figure 9 require subformulae labels $L^a_{\mathcal{M},\emptyset}(\psi, j)$ to be calculated for various ψ and j.

Any cached values that exist for these are only used if they do *not* have the "unknown" value. Otherwise the labels are recalculated and the cached values are overwritten. Witnesses for ExistsExp, etc. are computed when needed based on the cached values for the previous state and the recursive definition in Section 5.4, and then these and the resulting label are cached. The process terminates with an up-to-date value for $L^a_{\mathcal{M},\emptyset}(\phi, i{+}1)$, but the labels for ϕ at earlier states have not been updated (where necessary). This algorithm could be extended to solve the *prefix update* online model checking problem by explicitly starting a new calculation of $L^a_{\mathcal{M},\emptyset}(\phi, j)$ for all earlier state indices j where the cached value is "unknown". However, this computation may be wasted if there is no possibility that the label could change due to the addition of state m_{i+1}. For example, if $\phi = (@_n q) \vee (\bigcirc p)$ then when adding state m_{i+1} only state m_i needs to have its label recomputed (due to the subformula $\bigcirc p$), unless n is true in m_{i+1}, in which case all previous states need to have their label recomputed. We therefore need a more refined way to determine when a label may need to be updated.

The basic idea is to track which future states the label of ϕ at a given state m_i may depend on (termed its "dependencies"). Then when a future state m_j is added, we recompute the label for ϕ at i if j is one of i's dependencies and the label for ϕ at i is "unknown".

In the remainder of this section we present a solution to the prefix update online model checking problem. Section 7.1 formalises the notion of "may depend on" and gives a method for computing dependencies. Then, in Section 7.2, an algorithm for performing online checking is given, and an example is given in Section 7.3. Finally, in Section 7.4, we return to updating U formulae, and sketch a possible optimisation.

7.1. Tracking Dependencies

Examining the definitions in Figure 9 we can see there are three types of formula with labelling functions that are directly affected by the truncation of the model:

- $L^a_{\mathcal{M},g}(@_n\phi, j) = \langle j : (\top, \bot)\rangle$ for any j if n is a nominal that is not in the model. Extending the model with a new state in which n holds will allow that label to be recalculated using the other clause for $L^a_{\mathcal{M},g}(@_n\phi, j)$.

- $L^a_{\mathcal{M},g}(\bigcirc\phi, j) = \langle j : (\top, \bot)\rangle$ if j is the last state in the model. Adding a new state allows the label to be recalculated using the other clause for $L^a_{\mathcal{M},g}(\bigcirc\phi, j)$.

- The formula for $L^a_{\mathcal{M},g}(\phi \mathsf{U} \psi, j)$ involves an indexed disjunction and an indexed conjunction bounded by $|\mathcal{M}|$. Adding a new state changes $|\mathcal{M}|$ and invalidates that computation.

Adding a new state to the model may also affect any formula's labels in earlier states if it has a *subformula* of one of the above three types. This is due to the recursive definition of $L^a_{\mathcal{M},g}$. Based on this recursive computation of labels, we derive a dependency function $R(\phi, j)$ that, given the top-level formula ϕ, indicates for a given state m_j the indices of the future states that the label of ϕ at m_j may depend on. If a future state index k is not in $R(\phi, j)$, then adding state m_k cannot affect the label of ϕ at m_j. Compared with computing $L^a_{\mathcal{M},g}(\phi, j)$, computing $R(\phi, j)$ is simpler since it involves manipulating sets of indices, rather than labels, and $R(\phi, j)$ is only computed once (with the exception of dependencies on future-referring nominals), whereas the label function $L^a_{\mathcal{M},g}(\phi, j)$ may be recomputed a number of times.

We can formalise the intuition behind the dependency function R as follows:

$$(\forall i, j)\ \ j > i \wedge (L^a_{\mathcal{M}^j,g}(\phi, i) \neq L^a_{\mathcal{M}^{j-1},g}(\phi, i)) \ \Rightarrow\ j \in R(\phi, i)$$

In other words, when adding state m_j makes a difference to the label computed for ϕ at state m_i, then j must be in $R(\phi, i)$. Note that the implication is only in one direction: if $R(\phi, i)$ is defined compositionally, it cannot make any assumptions about the subformulae of ϕ and will, in general, compute a safe estimate that is a superset of the real dependency set.

Observe that there are two types of dependencies: the label of ϕ at i can depend on the *valuation* of propositions at state m_j, or it can depend on the *existence* of state m_j without depending on the valuation at m_j. For example, consider $\bigcirc @_n p$ being labelled at state m_2. The semantics for $\bigcirc$ on finite paths specify that even if n refers to a past state, the truth of $\bigcirc @_n p$ is not determined until state m_3 is available. Thus, the label for $\bigcirc @_n p$ at state m_2 depends on the valuation of proposition p at the state designated by n (the first type of dependency); and also on the existence of state m_3 (the second type of dependency). Although there is a sense in which the second type of dependency is weaker (since it does not depend on the valuation), it is still a dependency: if labelling ϕ at i depends on the existence of state m_j, then adding state m_j is a reason for recomputing the label for ϕ at i. We thus do not distinguish between the two dependency types, and, in deriving $R(\phi, i)$ from the definition of $L^a_{\mathcal{M},g}(\phi, i)$ treat any recursive call to $L^a_{\mathcal{M},g}(\psi, j)$ as representing a dependency.

Additionally, there are two points to note regarding the definition of R, both relating to hybrid logic features. Firstly, in order to deal with the binding operator $\downarrow$ we define R in terms of an auxilliary function R_g, which is defined relative to a function g that maps variables to states. Secondly, in order to deal with formulae of the form $@_s\phi$ where the state corresponding to s has not yet been added (and hence we do not know its index), the value of $R_g(\phi, i)$ is a set which may contain not just natural numbers (indices of states) but also nominals n. In fact in this case there is also another issue: because we don't know which state n refers to, we cannot continue to work out what other states might be depended upon. For example, suppose that n is a nominal that will be true in state m_4, which is not yet available, and that $\phi = @_n \bigcirc p$. In order to determine the truth of ϕ we need to access state m_5. However, we cannot know this until we know that n refers to state m_4. We address this by recording that there is a dependency on n, without capturing further dependencies arising from $\bigcirc p$ (i.e. from the subformula of $@_n \bigcirc p$), and when the state referred to by n becomes available, we recompute the dependency.

We thus define the dependency function $R(\phi, i)$ as follows:

$$R(\phi, i) = \{j \mid j \in R_\emptyset(\phi, i) \wedge (j > i \vee j \notin \mathbb{N})\}$$

where $R_g(\phi, i)$ is defined in Figure 12. In other words, we compute $R_g(\phi, i)$ with an initially empty g, and we exclude from the result any integers $j \leq i$ (the condition "$j \notin \mathbb{N}$" simply includes nominals).

The definition of R_g is, as noted earlier, derived from the definition of $L^a_{\mathcal{M},g}(\phi, i)$. A few cases need some explanation. For propositions, $R_g(p, i) = \emptyset$ because the dependency on state m_i is noted when the evaluation first shifts to that state, so there is no need to record it a second time. For $@_s\phi$ the dependency set $R_g(@_s\phi, i)$ is the state designated by s as well as any dependencies relating to the subformula ϕ. However, if the state designated by s is not available yet (which can only be the case if s is a nominal), then we simply note a dependency on the nominal (as discussed above).

Regarding ExistsExp, ExistsFulf and ExistsViol, recall that these modalities are all "backwards looking", as they are all defined in terms of $\mathsf{Trunc}_\mathsf{S}$ and so do not rely on future states.

Note that the definition of $L^a_{\mathcal{M},g}(\phi \mathsf{U} \psi)$ requires labels for subformulae up to the (current) final state of the model (at index $|\mathcal{M}|$), while $R_g(\phi \mathsf{U} \psi)$ is the infinite set of *all* potential future dependencies. This is because whenever a new state is added, the value of $|\mathcal{M}|$ needed in the recomputation of $L^a_{\mathcal{M},g}(\phi \mathsf{U} \psi)$ increases, and

$$
\begin{aligned}
R_g(p, i) &= \emptyset \\
R_g(p(s), i) &= \emptyset \\
R_g(x, i) &= \emptyset \\
\\
R_g(\phi \wedge \psi, i) &= R_g(\phi, i) \cup R_g(\psi, i) \\
R_g(\phi \vee \psi, i) &= R_g(\phi, i) \cup R_g(\psi, i) \\
R_g(\neg\phi, i) &= R_g(\phi, i) \\
\\
R_g(@_s\phi, i) &= \begin{cases} \{j\} \cup R_g(\phi, j) & \text{if } \exists\ j \text{ s.t. } \{m_j\} = [V, g](s) \\ \{s\} & \text{otherwise } (s \text{ can only be a nominal}) \end{cases} \\
R_g(\downarrow x.\phi, i) &= R_{g[x \mapsto m_i]}(\phi, i) \\
R_g(\exists_{p(x)}\phi, i) &= \bigcup_{m_j \in V(p)(m_i)} R_{g[x \mapsto m_j]}(\phi, i) \\
\\
R_g(\bigcirc\phi, i) &= \{i + 1\} \cup R_g(\phi, i + 1) \\
R_g(\ominus\phi, i) &= \begin{cases} \{i - 1\} \cup R_g(\phi, i - 1) & \text{if } i > 1 \\ \emptyset & \text{if } i \leq 0 \end{cases} \\
R_g(\phi \,\mathsf{U}\, \psi, i) &= \{i, i + 1, \ldots\} \\
R_g(\phi \mathsf{S} \psi, i) &= \{1, 2, \ldots, i\} \cup R_g(\psi, 1) \cup \bigcup_{1 < j \leq i} \Big(R_g(\phi, j) \cup R_g(\psi, j)\Big) \\
\\
R_g(\mathsf{ExistsExp}(\lambda, \rho), i) &= \{1, 2, \ldots, i\} \\
R_g(\mathsf{ExistsFulf}(\lambda, \rho), i) &= \{1, 2, \ldots, i\} \\
R_g(\mathsf{ExistsViol}(\lambda, \rho), i) &= \{1, 2, \ldots, i\}
\end{aligned}
$$

Figure 12: Definition of the dependency function $R_g(\phi, i)$

therefore as long as states continue to be added, the label will potentially depend on those states. This means when computing R_g we must handle infinite sets. However, there are standard techniques for computing with infinite sets (e.g. representing them lazily). It is also possible to use a symbolic represention, such as $\geq i$, to represent the set $\{i, i+1, \ldots\}$, and to extend set operators to deal with this representation. In order to simplify our presentation we assume that the underlying implementation uses a suitable technique for computing with infinite sets, and present our algorithm in terms of infinite sets.

It may seem that the definition of $R_g(\phi \mathsf{U} \psi, i)$ should include additional terms corresponding to the recursive calls $L^a_{\mathcal{M},g}(\psi, k)$ (for $i \leq k \leq |\mathcal{M}|$) and $L^a_{\mathcal{M},g}(\phi, k)$ (for $i \leq j < k \leq |\mathcal{M}|$) appearing in the definition of $L^a_{\mathcal{M},g}(\phi \mathsf{U} \psi)$, e.g.:

$$R_g(\phi \mathsf{U} \psi, i) \;=\; \{i, i+1, \ldots\} \cup \bigcup_{j \geq i} (R_g(\phi, j) \cup R_g(\psi, j))$$

However, the term $\{i, i+1, \ldots\}$ captures these dependencies for states m_j ($j \geq i$). Since $R_g(\phi \mathsf{U} \psi, i)$ is a superset of $\{i, i+1, \ldots\}$, evaluating the recursive calls to R_g can only make a difference if they result in dependencies on states prior to i. We argue that they can never make a difference to the result of $R(\phi, i)$ (Theorem 3, in Appendix B) and so eliminate them, resulting in the simpler definition given in Figure 12.

It is interesting to consider how the dependencies relate to the form of ϕ. If ϕ contains no temporal or hybrid logic constructs then none of the previously labelled states (m_1 to m_i) can be affected by the addition of state m_{i+1}. If ϕ is of the form $\bigcirc\phi'$ where ϕ' contains no temporal or hybrid constructs then only state m_i will need to be recomputed when state m_{i+1} is added. If $\phi = @_n\phi'$ where n is true in state m_{i+1} then the label of *all* of the previous states will need to be recomputed (but the computation will be a simple copy), but if n does not hold in state m_{i+1} then none of the previous states need to be updated. If $\phi = \psi \mathsf{U} \chi$ (where ψ and χ contain no temporal or hybrid logic constructs) then *all* previous states need to be reconsidered.

7.2. Online Checking Algorithm

The online checking algorithm below manipulates a number of data structures. Firstly, a proposition valuation function, V, maps each proposition to the set of states in which it holds, or, for state referencing propositions, to a mapping from state indices to sets of state indices (as described in Section 3, except here states are represented by their indices). Secondly, a dependencies map, D, maps each

state index i to the set $R(\phi, i)$ (where ϕ is the top-level formula being labelled). As discussed in the previous section, this is a set of state indices and/or nominals. Thirdly, each formula has a cached labels map, which maps state indices to labels. We write ϕ.labels[i] to indicate the label of ϕ at i. In other words, each (sub)-formula has a (possibly sparse) map recording its labels at different states (instead of the matrix-like structures shown in Figures 6 and 8).

Initially D is empty, the labels are empty (i.e. ϕ.labels is empty for all ϕ) and V maps every proposition to an empty set (or, for state-referencing propositions, the empty mapping), i.e. $V[p] = \emptyset$ for any proposition p. When we add a new state we are given the state's index i, a set V_i of propositions that are true in the state, and a set V'_i of mappings for state-referencing propositions, i.e. $V'_i = \{p \mapsto V_i^p, \ldots\}$ where V_i^p is a set of prior state indices, such that $p(x)$ holds in state m_i iff $x \in V_i^p$. We then:

1. Deal with state m_i itself:
   (a) Update the valuation V with V_i and V'_i
   (b) Compute the set of dependencies $R(\phi, i)$, i.e. the (future) states the label of ϕ at state m_i may depend on, and store the indices of these states in $D[i]$
   (c) Compute and cache the label for ϕ at state m_i
2. Recompute the labels for earlier states m_j ($j < i$) for which the label is $\langle j : (\top, \bot)\rangle$:
   (a) We first check for dependencies on nominals: if nominal n is true in state m_i (i.e. $n \in V_i$) and a previous state m_j may depend on the state indicated by n (i.e. $n \in D[j]$) then, now that the state named by the nominal is available, we recompute the dependency set for j.
   (b) We next check for dependencies: for any previous state m_j, if it may depend on state m_i (i.e. $i \in D[j]$) then we recompute the label of ϕ at state m_j.

This process is defined precisely by the following procedure for adding a new state m_i, given its valuation V_i and V'_i. The algorithm uses Nom to denote the set of all nominals. Note that this algorithm doesn't check whether a nominal n is true in exactly one state, but this check can easily be added.

The algorithm makes use of an optimisation: when considering which existing states may need to have their label recomputed (step 2 above), we only need to consider states for which the label for ϕ is the "unknown" label. In order to avoid having to check all previous states' labels, we introduce a set U which is simply

a set of indices of all (previous) states which have the "unknown" label for ϕ, i.e. $U = \{i \mid \phi.\text{labels}[i] = \langle i : (\top, \bot)\rangle\}$. Initially U is the empty set.

Procedure addState(ϕ, i,V_i, V_i')
// 1. Deal with state m_i
for each $p \in V_i$ do: *// Update V with V_i*
    $V[p] \leftarrow V[p] \cup \{i\}$
for each $(p \mapsto V_i^p) \in V_i'$ do: *// Update V with V_i'*
    $V[p] \leftarrow V[p] \cup \{i \mapsto V_i^p\}$
$D[i] \leftarrow R(\phi, i)$ *// Compute i's dependencies*
$\phi.\text{labels}[i] \leftarrow L^a_{\mathcal{M},\emptyset}(\phi, i)$ *// Compute i's label for ϕ*
if $\phi.\text{labels}[i] = \langle i : (\top, \bot)\rangle$ then *// Update U*
    $U \leftarrow U \cup \{i\}$

// 2. Deal with other states that may depend on i
for each state index $j \in U$ do:
    if $\exists n \in \mathsf{Nom},\ n \in (D[j] \cap V_i)$ then *// Check for nominal dependencies*
        $D[j] \leftarrow R(\phi, j)$ *... and update dependency set for j*
    if $i \in D[j]$ then *// Check for dependencies*
        $\phi.\text{labels}[j] \leftarrow L^a_{\mathcal{M},\emptyset}(\phi, j)$ *... and recompute label for ϕ at j*
        if $\phi.\text{labels}[i] \neq \langle i : (\top, \bot)\rangle$ then *// Update U*
            $U \leftarrow U \setminus \{i\}$

Theorem 2. *The preceding algorithm is correct. Formally, given a sequence of states $m_1, \ldots, m_k$, labelling them all at once using $L^a_{\mathcal{M},g}$ and adding them one-at-a-time using addState(ϕ, i,V_i, V_i') yields the same labelling.*
Proof (sketch): *The addState procedure uses $L^a_{\mathcal{M},g}$ to do the labelling of individual states. The only situation where a difference in labelling could arise is where the labelling function $L^a_{\mathcal{M},g}(\phi, i)$ is called initially when state m_i is added, and then is not recomputed when pertinent further information becomes available (i.e. additional future states are added). We thus establish correctness by arguing that labels are recomputed whenever there might be a need to do so.*

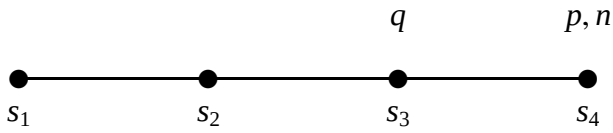
Key to this argument is the observation that the dependency function, $R(\phi, i)$, stored in D, is safe*: if $i \notin D[j]$ then the label of ϕ at j cannot depend on state m_i, and therefore there is no need to recompute the label for ϕ at j when state m_i is added. The slight complication is that if a nominal n is in the dependency set for j (i.e. $n \in D[j]$) then the set of dependencies may be incomplete. However, in this case once the state corresponding to n is added, the dependencies are recomputed, and at that point they become complete.*

The other case where the label is not recomputed is when state m_j depends on (subsequent) state m_i, but when i is added the label for ϕ at j is not "unknown" (i.e. $\langle j:(\top,\bot)\rangle$). To see that there is no need to recompute the label for ϕ at j we show that once the label of a state is no longer of the form $\langle j:(\top,\bot)\rangle$ then additional information (i.e. future states) will not affect the label for that state. There are two cases. Case 1: If the label is of the form $\langle j:(x,x)\rangle$ for $x \in \{\top,\bot\}$ then this is obvious, since the label indicates that ϕ is definitely true (if $x = \top$) or definitely false ($x = \bot$) at state m_j. Case 2: the label is of the form $\langle j:(\top,\bot), k:(x,x)\rangle$ where $k > j$. In this case, observe that states are added sequentially, and that, in order for this label to have been computed, state m_k must have already been added, and adding any subsequent states m_l (where $l > k$) can therefore have no effect on the label for ϕ at state m_j.

7.3. Example

To illustrate this process, we consider the formula $\phi = (@_n p) \mathsf{U} \bigcirc q$. This formula is not one that we would expect to be used, but it does allow various cases to be illustrated.

We consider a model in which the nominal n refers to state s_4, and where proposition q holds in state s_3, and proposition p holds in state s_4:



Initially $D = \emptyset$, ϕ.labels is empty for all ϕ, and $V[p] = V[q] = V[n] = \emptyset$. We now make the first state available, using addState(ϕ,1,$\emptyset$, $\emptyset$), i.e. none of the three propositions are true in the first state. Informally, we cannot determine the truth of $@_n p$ (because the state designated by the nominal n does not yet exist), nor of $\bigcirc q$ (because there is no next state). We therefore can only label s_1 with $\langle 1:(\top,\bot)\rangle$. Formally, addState($\phi$, 1, $\emptyset$, $\emptyset$) first computes $R(\phi, 1)$ and stores the result in $D[1]$. We compute $R(\phi, 1)$ by computing $R_\emptyset(\phi, 1)$. Since $\phi = (@_n p) \mathsf{U} \bigcirc q$ we have that $R_\emptyset(\phi, 1) = \{1, 2, \ldots\}$, and hence $R(\phi, 1) = \{2, 3, \ldots\}$, i.e. the label for ϕ at state s_1 may depend on all future states. Finally, addState(ϕ, 1, $\emptyset$, $\emptyset$) labels ϕ at state 1, concluding that ϕ.labels[1] = $\langle 1:(\top,\bot)\rangle$.

Now the second state, s_2, becomes available. We need to update V (no change, since no propositions are true in state 2), compute dependencies for state s_2, compute a label for it, and also re-compute the label for s_1 (since $2 \in D[1]$). The label

for s_2 is $\langle 2:(\top,\bot)\rangle$ (as per the label of s_1). We also compute $D[2]$ which, following analogous reasoning to that outlined in the previous paragraph, is $\{3,\ldots\}$. We then re-compute the label for s_1; however, it is unchanged: although we now know that $\bigcirc q$ is false in s_1, we cannot draw any conclusions about $(@_n p)\,\mathsf{U}\,\bigcirc q$ because it depends on whether $@_n p$ is true in s_1, which is still unknown.

The third state, s_3, then becomes available. We update V such that $V[q] = \{3\}$ (since q is true in state s_3), and compute the dependencies, resulting (as per previous discussion) in $D[3] = \{4,\ldots\}$. As before, $@_n p$ and $\bigcirc q$ are unknown, and so the label for s_3 is $\langle 3:(\top,\bot)\rangle$. We re-compute the labels for the first and second states. The label for s_1 is unchanged, However, we can now conclude that q is true in s_3 and that $\bigcirc q$ is true in s_2. This allows us to conclude that $(@_n p)\,\mathsf{U}\,\bigcirc q$ is true in s_2, and so the label of ϕ at s_2 is updated to $\langle 2:(\top,\bot), 3:(\top,\top)\rangle$, indicating that once state s_3 is available, we can conclude that ϕ holds at state s_2.

Finally the fourth state, s_4, becomes available. We update V such that $V[p] = \{4\}$ and $V[n] = \{4\}$. We also compute the dependency $D[4]$, which, as per earlier discussion, is $\{5,\ldots\}$. The label for ϕ at state s_4 is just $\langle 4:(\top,\bot)\rangle$, because we cannot conclude anything about $\bigcirc q$ in this state. We next re-compute the labels for s_1 and s_3 (but not s_2, since its label is no longer "unknown"). For s_3 the label is unchanged: although we now know that $\bigcirc q$ is false in s_3, we also now know that $@_n p$ is true, and so $(@_n p)\,\mathsf{U}\,\bigcirc q$ may be true if q is true in a future state. Considering s_1, state s_4 finally provides information about $@_n p$, and specifically $L_{\mathcal{M},g}(@_n p, 1) = \pi_1(L_{\mathcal{M},g}(p, 4)) = \langle 1:(\top,\bot), 4:(\top,\top)\rangle$. This can be used, with the definition of the label function for U, to derive the new label for ϕ at s_1: $\langle 1:(\top,\bot), 4:(\top,\top)\rangle$. Informally, once state s_4 is available we know that $@_n p$ is always true, which effectively reduces $(@_n p)\,\mathsf{U}\,\bigcirc q$ to $\Diamond\bigcirc q$, which holds in state s_1.

The following table summarises the labels. Each state has two labels: an "initial label" which is the label that is computed when the state is first added (and future states are not yet available), and the "final label" which is computed later, when sufficient information is available.

State	Propositions	Initial Label for ϕ	Final Label for ϕ
s_1	$\{\}$	$\langle 1:(\top,\bot)\rangle$	$\langle 1:(\top,\bot),\ 4:(\top,\top)\rangle$
s_2	$\{\}$	$\langle 2:(\top,\bot)\rangle$	$\langle 2:(\top,\bot),\ 3:(\top,\top)\rangle$
s_3	$\{q\}$	$\langle 3:(\top,\bot)\rangle$	(label doesn't change)
s_4	$\{p,n\}$	$\langle 4:(\top,\bot)$	$\langle 4:(\top,\bot)$

This example has illustrated how the addition of states can lead to the labels

for previous states being updated.

7.4. Optimising the update of U *formulae*

Use of the dependency function R reduces some unnecessary updates of cached labels, but it does not help with U formulae as the labels of these have (potential) dependencies on all future states. We now outline a technique that would help optimise the relabelling of $\phi \mathsf{U} \psi$.

Consider a label $l_{i,n} = L^a_{\mathcal{M}^n,g}(\phi \mathsf{U} \psi, i)$ that was cached when state m_n ($n > i$) was added to the model, i.e. when $|\mathcal{M}| = n$. The formula for this label shown in Figure 9 consists of two parts. The first line in the definition expresses the standard definition of U. The second line corresponds to the semantics of the "weak until" operator (sometimes denoted W) but is modified so that the strongest value it can contribute to the disjunction of the two lines is the "unknown" value $\langle i : (\top, \bot) \rangle$. In other words, this line allows the overall label to be "unknown" if ϕ is never known to be false from i onwards in the truncated model—it optimistically assumes that ψ will be true immediately after the truncation point. Note that both lines include a conjunction of the labels for ϕ over a range of states from m_i onwards. We define:

$$e(i,h) = \bigwedge_{i \le j \le h}^{i:\langle \top,\top \rangle} \pi_i(L^a_{\mathcal{M},g}(\phi, j))$$

Then line 1 of the equation for $l_{i,n}$ includes $e(i, k-1)$ (if we rewrite the upper limit of the summation to be defined using $\le$ rather than $<$) and line 2 includes $e(i, n)$.

Suppose that $l_{i,n}$ has the "unknown" value. When a new state m_{n+1} is added to the model, we need to update the cached label for ϕ at i to the new value $l_{i,n+1}$, preferably without computing it from scratch. The recalculation needed could be reduced if we collect and cache additional information during the labelling of U formulae. For example, while computing $l_{i,n}$ we could cache the maximum h for which $e(i, h)$ is not "false" (which we denote h^i_{max}). This value represents the maximum state in which ϕ possibly holds following m_i (based on the currently available future states). If $h^i_{max} < n$ then $e(i, n)$ must be "false" and so when computing $l_{i,n+1}$, $e(i, n{+}1)$ will also be "false". We can therefore ignore the second line in future recalculations of this label as more states are added. Furthermore, in the first line, $e(i, k{-}1)$ will be "false" for $k{-}1 > h^i_{max}$ so we do not need to add new disjuncts to the formula for $l_{i,n}$ as n increases. The value of the label can then only change as states are added to the model if the label for ϕ changes in any of the states m_j for $i \le j \le h^i_{max}$ or the label for ψ changes in any of the states m_j for $i \le j \le h^i_{max}{+}1$.

Thus, caching h^i_{max} allows the unnecessary recalculation of $l_{i,n}$ to be avoided in some situations. However, this value must still be stored separately for each i. An alternative approach would be to associate with each U formula ϕ two data structures recording (a) the maximal intervals $[i_1, i_2]$ of state indices for which ϕ remains at least "unknown", and (b) the set of state indices for which ψ is at least "unknown". This data would need to be updated as the model is extended and some "unknown" values become "false"—the "ϕ intervals" may shrink or split into separated subintervals, and the "ψ set" may get smaller. However, it would allow the formula for updating labels $l_{i,n}$ to be constrained in a similar way as for h^i_{max}: the outer disjunct would only need to consider ψ for states m_k that are elements of the "ψ set" and for which $[i, k-1]$ is a subset of some "ϕ interval". Further investigation of this approach is left for future work.

8. Related Work

There have been a variety of approaches to modelling expectations and commitments formally, some of which are outlined below.

Alberti *et al.* [21] represent conditional expectations as rules with an E modality in their conclusion. Abductive inference is used to generate expectations, which are monitored at run time. The temporal aspects of expectations are restricted to constraint logic programming constraints relating variables representing the time of events.

Verdicchio and Colombetti [22] use a first order variant of CTL* with past-time operators to provide axioms defining the lifecycle of commitments in terms of primitives representing the existence, the fulfilment, and the violation of a commitment in a state. In their approach, commitments are always expressed from the viewpoint of the state in which they were created, and the formula $Comm(e, a, b, u)$, recording that event e created a commitment from a to b that u holds, remains true in exactly that form from one state to the next. Fulfilment is then defined by a temporal formula that searches back in time for the event that created the commitment, and then evaluates the content u at that prior state, for all paths passing through the current state.

Bentahar *et al.* [23] present a logical model for commitments based on a branching time temporal logic in the context of semantics for argumentation. The semantics use accessibility relations for different types of commitments. These encode deadlines that are associated with commitments on their creation.

Model checking has been applied to statically verifying properties of closed systems of agents (thus their programs are available to form the input model) and

also for checking desired properties of institution specifications. An example of the latter is the work of Viganò and Colombetti [24]. Of more relevance to this paper is the application of model checking to run-time compliance checking based on observed traces.

Endriss [25] investigated the problem of checking whether a trace of an agent dialogue conforms to a protocol expressed in propositional linear temporal logic, by using *generalised model checking*: deciding whether there is any *extension* of the model that satisfies the protocol. This allows incomplete dialogues to be considered valid as long as they are following the protocol. He discussed the encoding of "future obligations" that are created by protocol rules, but in contrast to our semantic account of expectations, his approach relied on specific syntactical constructs to be used in the protocol definition (the use of a weak until operator and a special proposition *PENDING*, representing the existence of an unfulfilled obligation).

Bauer *et al.* [13] use standard, and well-known, automata-theoretic techniques to translate an LTL formula into a finite state machine that monitors an LTL formula by taking as input a finite trace, and, as each symbol is provided, produces as output either true, false, or unknown. One drawback of this approach is that instead of interpreting an LTL formula directly, it translates it into an automaton, which is potentially time consuming since the finite state machine can have size that is doubly exponential ($O(2^{2^n})$) in the size of the LTL formula. Since we are interested in applications where LTL formulae are specified at run-time by users [26], this costly compilation step is not desirable. Additionally, the logic they use (LTL) is more limited than what we use, since it excludes hybrid logic features, and also does not include past-time operators ($\ominus$ and S).

Spoletini and Verdicchio [27] addressed the online monitoring of commitments expressed in a propositional temporal logic with both past and future operators. The formula to be monitored is preprocessed by applying Gabbay's rules [28] to separate out past-oriented formulae that are nested within future-oriented formulae so that these occur in separate subformulae. Due to the non-elementary blow-up in formula complexity that can result from this procedure, it is assumed that formulae are already separated or have only a small number of such nestings. The past-directed formulae are then treated as propositions, the values of which are generated for each new event by deterministic Büchi automata generated from these formulae. The transformed formula (which now only contains propositions and future-oriented formulae) is then translated into an alternating modulo counting automaton that processes an input sequence of propositions and accepts or rejects it when it has been determined that the formula has become true or false

when evaluated on that sequence. This use of automata has the significant advantage of not requiring a history of observations to be kept—all required information from the history is captured by the states of the various automata. On the other hand, the current state of a formula being monitored (but not yet found to be true or false) cannot be tracked as in our framework, where the ExistsExp operator and its progression-based semantics give an explicit representation of the current (partially evaluated) expectations arising from a rule. It is also not clear how feasible it would be to extend this automata-based approach to more expressive forms of temporal logic.

Ågotnes *et al.* [29] extend temporal logic (specifically CTL) with the ability to model norms. They do this by extending CTL with obligation and prohibition operators of the form O_η (respectively P_η) . They define a "normative system" (designated η) simply as a set of transitions that are forbidden. In addition to defining a logic and giving it semantics and an axiomatisation, they investigate the complexity of model checking. It is worth noting that they are model checking over the whole Kripke structure, whereas we are concerned with model checking over (finite) traces, and that therefore their complexity results do not apply to our work. Another difference is that their logic does not include hybrid logic constructs.

9. Conclusions and Future Work

This paper has presented a logical account of the notions of conditional expectation, fulfilment and violation in terms of a linear temporal logic. For offline monitoring of expectations, the problem of determining fulfilment and violation of expectations without recourse to future information was identified as a key problem, and a solution was presented in terms of path truncation and the strong semantics of Eisner *et al.* [12]. It was then shown how the MCFULL model checking algorithm can be modified to support the truncation operator by using generalised labels that record for a model state the truth values under both the weak and strong semantics for all possible future states. An existing model checker (HLMC) has been modified using these techniques to allow the existence of expectations, and fulfilments and violations of these expectations to be detected, and much more concise implementation in Python has also been developed as a basis for future extensions. Two versions of the online model checking problem were also defined, and an algorithm for online model checking was presented, based on an analysis of when previously computed labels may need to be updated as new states are added to the model.

Including nominals in our logic allowed our Exp modality to record the states in which expectations were created. The Second Life example in Section 6.2 also demonstrated the use of the other hybrid logic features in conjunction with a state-referencing proposition. We plan to extend our approach to apply to a real-time hybrid temporal logic interpreted over timed paths. In this case, the bind operator and state variables become useful for expressing timing relations between states. We also plan to investigate extending the technique to apply to some suitably constrained fragment of first order temporal logic (e.g. the loosely guarded fragment). In this case, it would be useful if nominals with particular meaning to the application could appear within atomic formulae in the model.

An issue relating to the "end of time" is that when we know that an execution is over, and that there will be no further states, then we want to be able to draw stronger conclusions, such as concluding that an expectation $\Diamond p$ is violated if p has not yet occurred, or that an expectation $\Box p$ is fulfilled if p has always occurred. Extending the semantics to deal with this is a subject for future work.

Other future avenues for work include investigating the representation of interdependent expectations, and exploring techniques for representing the state history in a more compact way (e.g. by annotating propositions with the maximal intervals of time in which they hold [30]).

This paper focused on the logical and algorithmic details of our approach to expectation monitoring. There are also a number of pragmatic issues to be addressed when applying our technique to a real scenario. These include providing a flexible way to map (only) the relevant real-world events to propositions, allowing clients of the system to specify notification policies for finer grained control of the messages they receive from the model checker [6], and developing techniques for eliciting rules of expectation from non-expert human users [26]. Other pragmatic issues, possible limitations and desirable additional features will come to light from exploring a range of applications, such as the use of the model checker to monitor user's or communities' rules of expectation in the Second Life virtual world [17].

Acknowledgements
We would like to thank Surangika Ranathunga for her work on monitoring events in Second Life as outlined in Section 6.2, and the anonymous reviewers for their comments which helped to improve this paper.

References

[1] U. Cortés, Electronic institutions and agents, AgentLink News 15 (2004) 14–15.

[2] L. Dragone, Hybrid logics model checker, http://luigidragone.com/hlmc/ (2005).

[3] S. Cranefield, A rule language for modelling and monitoring social expectations in multi-agent systems, in: O. Boissier, J. A. Padget, V. Dignum, G. Lindemann, E. T. Matson, S. Ossowski, J. S. Sichman, J. Vázquez-Salceda (Eds.), Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems, Vol. 3913 of Lecture Notes in Artificial Intelligence, Springer, 2006, pp. 246–258.

[4] N. Markey, P. Schnoebelen, Model checking a path (preliminary report), in: R. M. Amadio, D. Lugiez (Eds.), CONCUR 2003 – Concurrency Theory, Vol. 2761 of Lecture Notes in Computer Science, Springer, 2003, pp. 251–265.

[5] S. Cranefield, M. Winikoff, Verifying social expectations by model checking truncated paths, in: J. Hübner, E. Matson, O. Boissier, V. Dignum (Eds.), Coordination, Organizations, Institutions and Norms in Agent Systems IV, Vol. 5428 of Lecture Notes in Artificial Intelligence, Springer, 2009, pp. 204–219.

[6] S. Cranefield, Modelling and monitoring social expectations in multi-agent systems, in: P. Noriega, J. Vázquez-Salceda, G. Boella, O. Boissier, V. Dignum, N. Fornara, E. Matson (Eds.), Coordination, Organizations, Institutions, and Norms in Agent Systems II, Vol. 4386 of Lecture Notes in Artificial Intelligence, Springer, 2007, pp. 308–321.

[7] F. Bacchus, F. Kabanza, Using temporal logics to express search control knowledge for planning, Artificial Intelligence 116 (1-2) (2000) 123–191.

[8] M. Franceschet, M. de Rijke, Model checking hybrid logics (with an application to semistructured data), Journal of Applied Logic 4 (3) (2006) 279–304.

[9] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, On the temporal analysis of fairness, in: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on principles of programming languages, ACM, 1980, pp. 163–173.

[10] O. Lichtenstein, A. Pnueli, L. Zuck, The glory of the past, in: R. Parikh (Ed.), Logics of Programs, Vol. 193 of Lecture Notes in Computer Science, Springer, 1985, pp. 196–218.

[11] M. Pradella, P. San Pietro, P. Spoletini, A. Morzenti, Practical model checking of LTL with past, in: Proceedings of the 1st International Workshop on Automated Technology for Verification and Analysis, 2003, http://cc.ee.ntu.edu.tw/~atva03/papers/13.pdf.

[12] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, D. V. Campenhout, Reasoning with temporal logic on truncated paths, in: W. A. Hunt, Jr., F. Somenzi (Eds.), Computer Aided Verification, Vol. 2725 of Lecture Notes in Computer Science, Springer, 2003, pp. 27–39.

[13] A. Bauer, M. Leucker, C. Schallhart, Monitoring of real-time properties, in: S. Arun-Kumar, N. Garg (Eds.), FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, Vol. 4337 of Lecture Notes in Computer Science, Springer, 2006, pp. 260–272.

[14] J. Snell, Automating business processes and transactions in Web services, http://www.ibm.com/developerworks/webservices/library/ws-autobp/ (2002).

[15] W. M. P. van der Aalst, M. Pesic, Decserflow: Towards a truly declarative service flow language, in: M. Bravetti, M. Núñez, G. Zavattaro (Eds.), Web Services and Formal Methods, Vol. 4184 of Lecture Notes in Computer Science, Springer, 2006, pp. 1–23.

[16] Linden Lab, Second Life home page, http://secondlife.com/ (2008).

[17] S. Cranefield, G. Li, Monitoring social expectations in Second Life, in: J. Padget, A. Artikis, W. Vasconcelos, K. Stathis, V. Silva, E. Matson, A. Polleres (Eds.), Coordination, Organizations, Institutions and Norms in Agent Systems V, Vol. 6069 of Lecture Notes in Artificial Intelligence, Springer, 2010, pp. 133–146.

[18] Vstex Company, Second Life Football System, http://www.secondfootball.com/ (2008).

[19] Open Metaverse Foundation, LibOpenMetaverse, http://www.openmetaverse.org/projects/libopenmetaverse (2009).

[20] EsperTech Inc., Esper – Complex Event Processing, http://esper.codehaus.org/ (2006).

[21] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, P. Torroni, Compliance verification of agent interaction: a logic-based software tool, in: R. Trappl (Ed.), Cybernetics and Systems 2004, Vol. II, Austrian Society for Cybernetics Studies, 2004, pp. 570–575.

[22] M. Verdicchio, M. Colombetti, A logical model of social commitment for agent communication, in: Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003), ACM Press, 2003, pp. 528–535.

[23] J. Bentahar, B. Moulin, J.-J. C. Meyer, B. Chaib-draa, A logical model for commitment and argument network for agent communication, in: Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), IEEE Computer Society, 2004, pp. 792–799.

[24] F. Viganò, M. Colombetti, Symbolic model checking of institutions, in: Proceedings of the 9th International Conference on Electronic Commerce, ACM Press, 2007, pp. 35–44.

[25] U. Endriss, Temporal logics for representing agent communication protocols, in: Agent Communication II, Vol. 3859 of Lecture Notes in Artificial Intelligence, Springer, 2006, pp. 15–29.

[26] M. Winikoff, S. Cranefield, Eliciting expectations for monitoring social interactions, in: M. Purvis, B. T. R. Savarimuthu (Eds.), Computer-Mediated Social Networking, Vol. 5322 of Lecture Notes in Artificial Intelligence, Springer, 2009, pp. 171–185.

[27] P. Spoletini, M. Verdicchio, An automata-based monitoring technique for commitment-based multi-agent systems, in: J. Hubner, E. Matson, O. Boissier, V. Dignum (Eds.), Coordination, Organizations, Institutions and Norms in Agent Systems IV, Vol. 5428 of Lecture Notes in Computer Science, Springer, 2009, pp. 172–187. doi:10.1007/978-3-642-00443-8_14.

[28] D. M. Gabbay, The declarative past and imperative future: Executable temporal logic for interactive systems, in: B. Banieqbal, A. Pnueli, H. Barringer

(Eds.), Temporal Logic in Specification, Vol. 398 of Lecture Notes in Computer Science, Springer, 1989, pp. 409–448.

[29] T. Ågotnes, W. van der Hoek, J. A. Rodríguez-Aguilar, C. Sierra, M. Wooldridge, On the logic of normative systems, in: M. Veloso (Ed.), Proceedings of the 20th International Joint Conference on Artificial Intelligence, AAAI Press, 2007, pp. 1175–1180.

[30] M. Fisher, R. Owens, From the past to the future: Executing temporal logic programs, in: A. Voronkov (Ed.), Logic Programming and Automated Reasoning, Vol. 624 of Lecture Notes in Computer Science, Springer, 1992, pp. 369–380.

Appendix A. Proof of Theorem 1: $L^a_{\mathcal{M},g}(\phi, i)$ produces simple labels

In this appendix we show that our labelling function $L^a_{\mathcal{M},g}$ correctly produces only simple labels. First we show that if l is a simple label, then so are $l \downharpoonright k$ and $\pi_k\, l$ (both operators are defined in Figure 9).

Lemma 1. *If l is simple then so is $l \downharpoonright k$*
Proof: *Consider each of the three forms. Form 1: $l = \langle i\!:\!(x,x)\rangle$. If $k > i$ then $l \downharpoonright k = \langle k\!:\!(x,x)\rangle$ which is simple. If $k \leq i$ then $l \downharpoonright k = l$ which is simple (note that this case doesn't actually arise in Figure 9). Form 2: $l = \langle i\!:\!(\top,\bot), j\!:\!(x,x)\rangle$. If $i < k < j$ then $l \downharpoonright k$ is just l with i replaced by k (and if $k \leq i$ then, as in the previous case, the result is just l). If $j \leq k$ then $l \downharpoonright k$ is just $\langle k\!:\!(x,x)\rangle$ where x is the value associated with j in l. In either case the result is a simple label. For form 3, $l = \langle i\!:\!(\top,\bot)\rangle$, the argument is similar to that of form 1.* □

Lemma 2. *If l is simple then so is $\pi_k l$.*
Proof: *There are two cases for $\pi_k l$. In the second case $\pi_k l = l \downharpoonright k$ which, by lemma 1, is simple. In the first case $\pi_k l = \langle k\!:\!(\top,\bot)\rangle \bullet l$. If l is of the form $\langle i\!:\!(x,x)\rangle$ then the result is simple. If l is of the form $\langle i\!:\!(\top,\bot), j\!:\!(x,x)\rangle$ or $\langle i\!:\!(\top,\bot)\rangle$ then we observe that $i > k$ (by the condition on the first case of π_k) and hence that prepending $k\!:\!(\top,\bot)$ just replaces the i with a k, that is, $\langle k\!:\!(\top,\bot), i\!:\!(\top,\bot), \ldots\rangle$ is abbreviated to $\langle k\!:\!(\top,\bot), \ldots\rangle$* □

Proof of Theorem 1: $L^a_{\mathcal{M},g}(\phi, i)$ produces simple labels. The proof is by induction over the structure of ϕ. In the base cases (p, $p(s)$, x, $\top$, and $\bot$) the labels are clearly simple. For $@_s$ we argue that the label is simple because the label of the sub-formula (l_1) is simple (by the induction hypothesis), and hence $\pi_i l_1$ is also simple (by lemma 2). A similar argument applies for $\bigcirc$ and $\ominus$. For $\downarrow x$ we merely have to apply the induction hypothesis. Negation changes the values associated with the labels, but not the structure of the labels, and so we need to only note that $\neg(\top,\bot) = (\top,\bot)$ (since negation swaps weak and strong and then negates) to show that negating a simple label results in a simple label.

Let us now consider conjunction and let $l = L^a_{\mathcal{M},g}(\phi \wedge \psi, i)$ which is just $L^a_{\mathcal{M},g}(\phi, i) \wedge L^a_{\mathcal{M},g}(\psi, i)$. Suppose that the labels of both sub-formulae are length 1, then they both have labels of the form $\langle i\!:\!(y,z)\rangle$ (where $(y,z) = (\top,\bot)$ or $y = z$). We therefore can easily see that l has the form $\langle i\!:\!(y,z)\rangle$, and that possible values for (y,z) are one of $(x \wedge y, x \wedge y)$ (if both labels are in the first form), or $(\top,\bot)$ (if both labels are in the third form), or $(\top \wedge x, \bot \wedge x)$ (if one label is first form and the other third form). In this last case, either $x = \top$ in which case the value is $(\top,\bot)$

which gives a simple label; or $x = \bot$, in which case the value is $(\bot, \bot)$ which also gives a simple label. For disjunction, in this last case, the value is $(\top \vee x, \bot \vee x)$ which is just $(\top, x)$ which results in a simple label for $x = \top$ or $x = \bot$.

We now consider the case for conjunction where the labels of the two subformulae are in the second form. Let $l_1 = L^a_{\mathcal{M},g}(\phi, i)$ which, by induction hypothesis and the assumption of second form, can be written as $l_1 = \langle i\!:\!(\top, \bot), j\!:\!(x, x)\rangle$. Similarly, we have $l_2 = L^a_{\mathcal{M},g}(\psi, i)$ which can be written as $l_2 = \langle i\!:\!(\top, \bot), k\!:\!(y, y)\rangle$. If $j = k$ then it is trivial to see that l is simple, and so we now consider $j \neq k$ and, without loss of generality, assume that $j < k$. Now, by the definition of conjunction over labels, we have that: $l = \langle i\!:\!(\top, \bot), j\!:\!(x \wedge \top, x \wedge \bot), k\!:\!(x \wedge y, x \wedge y)\rangle$.

If $x = \top$ then this is just $\langle i\!:\!(\top, \bot), j\!:\!(\top, \bot), k\!:\!(\top \wedge y, \top \wedge y)\rangle$ which can be simplified to $\langle i\!:\!(\top, \bot), k\!:\!(y, y)\rangle$ which is simple. If $x = \bot$ then this is just $\langle i\!:\!(\top, \bot), j\!:\!(\bot, \bot), k\!:\!(\bot \wedge y, \bot \wedge y)\rangle$ which is just $\langle i\!:\!(\top, \bot), j\!:\!(\bot, \bot), k\!:\!(\bot, \bot)\rangle$ and can be simplified to $\langle i\!:\!(\top, \bot), j\!:\!(\bot, \bot)\rangle$ which is simple. For disjunction we have that if $x = \top$ then l is $\langle i\!:\!(\top, \bot), j\!:\!(\top, \top), k\!:\!(\top \vee y, \top \vee y)\rangle$ which can be simplified to the simple label $\langle i\!:\!(\top, \bot), j\!:\!(\top, \top)\rangle$; and if $x = \bot$ then l is $\langle i\!:\!(\top, \bot), j\!:\!(\top, \bot), k\!:\!(\bot \vee y, \bot \vee y)\rangle$ which can be simplified to the simple label $\langle i\!:\!(\top, \bot), k\!:\!(y, y)\rangle$. Thus in all cases for conjunction and disjunction the resulting label is simple.

For $\exists_{p(x)}\phi$ we observe that by the induction hypothesis the label for ϕ is simple. Since the label for $\exists_{p(x)}\phi$ is therefore a disjunction of simple labels, it is also simple.

Finally, we consider U and S. Since these are defined in terms of conjunction and disjunction, it follows that the resulting labels here are also simple. □

Appendix B. Recursive calls are not needed to compute $R_g(\phi \mathsf{U} \psi, 1)$

The following theorem shows that the definition of $R_g(\phi \mathsf{U} \psi, i)$ can be simplified by omitting recursive calls to $R_g(\phi, j)$ and $R_g(\psi, j)$.

Theorem 3. *Define R'_g as being identical to R_g except that $R'_g(\phi \mathsf{U} \psi, i) = \{i, i+1, \ldots\} \cup \bigcup_{j \geq i}(R'_g(\phi, j) \cup R'_g(\psi, j)$. Then define $R'(\phi, i) = \{j \mid j \in R'_\emptyset(\phi, i) \wedge (j > i \vee j \notin \mathbb{N})\}$. For any ϕ and any i we have that $R(\phi, i) = R'(\phi, i)$.*

Proof (sketch): *Consider the process of computing $R(\phi, i)$ and $R'(\phi, i)$. The only difference between them concerns (sub-)formulae of the form $\psi_1 \mathsf{U} \psi_2$. How might this difference in definition result in $R(\phi, i) \neq R'(\phi, i)$? Since neither $R(\phi, i)$ nor $R'(\phi, i)$ include states $\leq i$, in order for there to be a difference we must have a "gap", specifically:*

- *$\psi_1 \cup \psi_2$ must be a sub-formula of ϕ (if $\phi = \psi_1 \cup \psi_2$ then $R(\phi, i) = R'(\phi, i) = \{i, i+1, \ldots\}$); and*
- *in order to evaluate $R(\phi, i)$ (resp. $R'(\phi, i)$) we evaluate $R_g(\psi_1 \cup \psi_2, j)$ (resp. $R'_g(\psi_1 \cup \psi_2, j)$ where $i < j$; and*
- *there exists k ($i < k < j$ such that $k \notin R_g(\phi, i)$ but $k \in R'_g(\phi, i)$.*

Observe that these conditions correspond to the existence of a "gap" in $R(\phi, i)$ where $R(\phi, i)$ includes $\{j, j+1, \ldots\}$ ($j > i$) but, excludes some k between i and j. In order for this to occur we need to begin with $R(\phi, i)$ and end with a recursive call to $R_g(\psi_1 \cup \psi_2, j)$ (where $j > i$), without including k in the result. There are only two types of formula that could potentially cause such a recursion:

1. *$@_s\phi$: Computing the dependencies of $@_s\phi$ at i requires knowing the dependencies of ϕ in the state m_j referred to by s, which could be in the future of state m_i. However, if s is a future-referring nominal n, then we simply hold off until n arrives (i.e. we do not immediately evaluate $R_g(\phi, j)$). Once the state named by n arrives the incremental algorithm model checking algorithm will be run starting at that state, so there will no longer be a potential "gap" in the states computed for $R(\phi, i)$.*
2. *$\exists_{p(x)}$: Since $p(x)$ is assumed to be backwards referring only, this cannot shift evaluation to the future.*

Therefore we cannot have a situation where $R(\phi, i)$ and $R'(\phi, i)$ differ.