METATEM:
A Framework for Programming in Temporal Logic *

Howard Barringer[†], Michael Fisher[†], Dov Gabbay[‡],
Graham Gough[†] and Richard Owens[‡]

[†] Department of Computer Science,
University of Manchester,
Oxford Road,
Manchester M13 9PL, UK.
{howard,michael,graham}@uk.ac.man.cs

[‡] Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate,
London SW7 2BZ, UK.
{dg,rpo}@uk.ac.ic.doc

Abstract. In this paper we further develop the methodology of temporal logic as an executable imperative language, presented by Moszkowski [Mos86] and Gabbay [Gab87, Gab89] and present a concrete framework, called METATEM for executing (modal and) temporal logics. Our approach is illustrated by the development of an execution mechanism for a propositional temporal logic and for a restricted first order temporal logic.

Keywords. Modal and Temporal Logics, Reactive Systems, Specification, Verification, Synthesis, Mechanical Verification, Prototyping, Rule-based Systems, Non-procedural Languages, Logic Programming.

Contents

*Work supported partially by Alvey under grants PRJ/SE/054 and IKBS/170 and by ESPRIT under Basic Research Action 3096 (SPEC).

1 Introduction

The purpose of this paper is twofold:

1. to re-emphasise and further develop the methodology of temporal logic as an executable imperative language, presented by Moszkowski [Mos86] and Gabbay [Gab87, Gab89]

2. to present a concrete framework, called METATEM for executing (modal and) temporal logics, illustrated by the development of an execution mechanism for a propositional temporal logic and then for restricted first order temporal logic.

The main import of this paper is the latter, and it is developed within the more general framework of the former. This methodology can serve as the natural meeting ground for the declarative and imperative approaches in computing, namely *imperative logic*.

We have structured this presentation on METATEM as follows. Section 1 outlines the METATEM approach and compares it to other work on the execution of temporal logics. Section 2 illustrates this approach by the introduction of a propositional temporal logic for METATEM. An execution mechanism for this form of METATEM is described and an outline given for an interpreter. Section 3 expands the description of the interpreter and discusses some of the issues raised in the its design. Section 4 introduces a limited first order temporal logic and describes its execution mechanism. Section 5 concludes the paper by discussing a general framework in which the METATEM approach to executing temporal logic resides.

1.1 Motivation

We distinguish two views of logic, the declarative and the imperative. The declarative view is the traditional one, and it manifests itself both syntactically and semantically. Syntactically a logical system is taken as being characterised by its set of theorems. It is not important how these theorems are generated. Two different algorithmic systems generating the same set of theorems are considered as producing the same logic. Semantically a logic is considered as a set of formulae valid in all models. The model $\mathcal{M}$ is a static semantic object. We evaluate a formula φ in a model and, if the result of the evaluation is positive (notation $\mathcal{M} \models \varphi$), the formula is valid. Thus the logic obtained is the set of all valid formulae in some class $\mathcal{K}$ of models.

In contrast to the above, the imperative view regards a logic syntactically as a dynamically generated set of theorems. Different generating systems may be considered as different logics. The way the theorems are generated are an integral part of the logic. From the semantic viewpoint, a logical formula is not *evaluated* in a model but performs *actions* on a model to get a *new* model. Formulae are accepted as valid according to what they do to models. For example we may take φ to be valid in $\mathcal{M}$ if $\varphi(\mathcal{M}) = \mathcal{M}$. (i.e. $\mathcal{M}$ is a fixed-point of φ.)

Applications of logic in computer science have mainly concentrated on the exploitation of its declarative features. Logic is taken as a language for describing properties of models. The formula φ is evaluated in a model $\mathcal{M}$. If φ holds in $\mathcal{M}$ (evaluation successful) then $\mathcal{M}$ has property φ. This view of logic is, for example, most suitably and most successfully exploited in the areas of databases and in program specification and verification. One can present the database as a deductive logical theory and query it using logical formulae. The logical evaluation process corresponds to the computational querying process. In program verification, for example, one can describe in logic the properties of the programs to be studied. The description plays the rôle of a model $\mathcal{M}$. One can now describe one's specification as a logical formula φ, and the query whether φ holds in $\mathcal{M}$ (denoted $\mathcal{M} \vdash \varphi$) amounts to verifying that the program satisfies the specification. These methodologies rely solely on the declarative nature of logic.

Logic programming as a discipline is also declarative. In fact it advertises itself as such. It is most successful in areas where the declarative component is dominant, for example deductive databases. Its procedural features are not imperative (in our sense) but computational. In the course of evaluating whether $\mathcal{M} \vdash \varphi$, a procedural reading of $\mathcal{M}$ and φ is used. φ does not imperatively act on $\mathcal{M}$, the declarative logical features are used to guide a procedure, that of taking steps for finding whether φ is true. What does not happen is that $\mathcal{M}$ and φ are read imperatively, resulting in some action. In logic programming such actions (e.g. assert) are obtained by side-effects and special non-logical imperative predicates and are considered undesirable. There is certainly no conceptual framework within logic programming for allowing only those actions which have logical meaning.

Some researchers have come close to touching upon the imperative reading of logic. Belnap [Bel77] and the later so-called data semantics school regard a formula φ as generating an action on a model $\mathcal{M}$, and changing it. In logic programming and deductive databases the handling of integrity constraints borders on the use of logic imperatively. Integrity constraints have to be maintained. Thus one can either reject an update or do some corrections. Maintaining integrity constraints is a form of executing logic, but it is logically ad-hoc and has to do with the local problem at hand. Truth maintenance is another form. In fact, under a suitable interpretation, one may view any resolution mechanism as model building which is a form of execution. In temporal logic, model construction can be interpreted as execution. Generating the model, i.e. finding the truth values of the atomic predicates in the various moments of time, can be taken as a sequence of execution.

As the need for the imperative executable features of logic is widespread in computer science, it is not surprising that various researchers have touched upon it in the course of their activity. However there has been no conceptual methodological recognition of the imperative paradigm in the community, nor has there been a systematic attempt to develop and bring this paradigm forward as a new and powerful logical approach in computing.

The area where the need for the imperative approach is most obvious and pressing is temporal logic. In general terms, a temporal model can be viewed as a progression of ordinary models. The ordinary models are what is true at each moment of time. The imperative view of logic on the other hand also involves step by step progression in virtual "time", involving both the syntactic generation of theorems and the semantic actions of a temporal formula on the temporal model. Can the two intuitive progressions, the semantic time and the action (transaction) time, be taken as the same? In the case of temporal logic the answer is yes. We can act upon the models in the same time order as their chronological time. This means acting on earlier models first. In fact intuitively a future logical statement can be read (as we shall see) both declaratively and imperatively. Declaratively it describes what should be true, and imperatively it describes the actions to be taken to ensure that it becomes true. Since the chronology of the action sequence and the model sequence are the same, we can virtually *create* the future model by our actions. The logic USF, defined by Gabbay [Gab89], was the first attempt at promoting the imperative view as a methodology, with a proposal for its use as a language for controlling processes.

1.2 The METATEM Approach

Work in the field of logic programming has demonstrated the utility of systems within which a single statement can be given both declarative and procedural readings. However, adapting existing systems to handle time dependent data and reasoning has proved to be difficult, since attempts to do so have involved simulating time rather than incorporating it at the heart of the system. Our approach is to place temporal considerations at the centre of the execution model, while maintaining the advantages of both declarative and procedural readings. A consequence of this is a more powerful expressive framework which incorporates Horn clause logic as a subset. Consider a temporal sentence of the form:

$$\text{antecedent about the past} \Rightarrow \text{consequent about the present and future}$$

this can, in general, be interpreted as if the "antecedent (about the past)" is true then **do** the "consequent (about the present and future)". Adopting this imperative reading, in fact, yields us with an execution mechanism for temporal logics. We take this as the basis of the METATEM approach. The name METATEM, in fact, captures two key aspects of our developing framework:

- the use of TEMporal logic as a vehicle for specification and modelling, via direct execution of the logic, of reactive systems;
- the direct embodiment of META-level reasoning and execution, in particular, identifying metalanguage and language as one and the same, providing a highly reflective system.

This paper focusses on the former point. Generally, the behaviour of a reactive component is given by specifying the interactions that occur between that component and the context, or environment, in which it is placed. In particular, a distinction needs to be made between actions made by the component and those made by the environment. In METATEM, we perpetuate such distinctions. The behaviour of a reactive component is described by a collection of temporal rules, in the form mentioned above. The occurrence, or otherwise, of an action is denoted by the truth, or falsity, of a proposition. The mechanism by which propositions are linked to actual action occurrences is not an issue here and is left to the reader's intuition. However, the temporal logics presented in Sections 2.1 and 4 do distinguish between component and environment propositions.

1.3 Related Work

The study of temporal and modal logics in the formal development of reactive systems has, in the main, been concentrated on techniques for specification and verification. The study of synthesis from temporal logic based specification, e.g. [MW84] and more recently [PR89a, PR89b], and, alternatively, the direct execution of temporal and modal logic, e.g. [Mos86] has not been so widespread, despite there being some fascinating starts.

Currently, the most developed implementation of an executable temporal logic is that of Moszkowski and Hale. Moszkowski [Mos86] has implemented a subset of a temporal logic, known as Tempura, based on interval models [MM84]. In fact, that particular interval temporal logic was first introduced for describing hardware [Mos83] and has now been put to the test with some examples of hardware simulation within Tempura [Hal87]. However, although Tempura is, in the computational sense, expressive enough, Tempura is limited in the sense of logic programming as in Prolog. Concurrent with the development of Moszkowski's work, Tang [Zhi83] defined and implemented a low-level temporal (sequential) programming system XYZ/E. His logic is based upon a linear, discrete, future-time temporal logic. Recent developments in that area have been developments of the logic (to obtain XYZ/D) to handle limited forms of distributed programming. Again, the drawback of this work, from the declarative and logic programming perspective, is the very traditional low-level state machine style. Steps towards the development of richer and more expressive executable subsets have since occurred. In particular, Gabbay [Gab87, Gab89] who describes general execution models, Abadi et al. [AM85, AM86, Aba87b, Aba87a, Bau89] who investigate temporal resolution mechanisms, Sakuragawa [Sak86] for a similar resolution-based "Temporal Prolog" and Hattori et al. [HNNT86] describe a modal logic based programming language for real-time process control, RACCO, based on Temporal Prolog.

2 Propositional METATEM and its Execution

In this section we illustrate the general principles outlined in Section 1 by the development of an execution mechanism for a propositional linear temporal logic.

2.1 A Propositional Linear-time Temporal Logic

We introduce a basic *linear* and *discrete* propositional temporal logic, PML (propositional METATEM logic). The language is obtained by augmenting classical propositional logic with *temporal* operators. The reader who is familiar with such standard temporal logics may prefer to skip the next two subsections and proceed to Section 2.2 where our execution approach is outlined.

The logic is presented in the usual way: Section 2.1.1 introduces the syntactic elements; Section 2.1.2 introduces the semantics.

2.1.1 Syntax

The basic symbols of the logic are depicted in Figure 1. We assume an alphabet of proposition symbols $\mathcal{A}_P$ which is the union of two disjoint sets $\mathcal{A}_C$ and $\mathcal{A}_E$ of propositions controlled, respectively, by the component and by the environment. A standard collection of propositional connectives together with a collection of temporal connectives complete the basic symbols. $\bigcirc$, $\square$ and $\lozenge$ are unary *future*-time connectives; $\mathcal{W}$ and $\mathcal{U}$ are binary *future*-time connectives; $\bullet$, $\blacksquare$ and $\blacklozenge$ are unary *past*-time connectives; $\mathcal{Z}$ and $\mathcal{S}$ are binary *past*-time connectives.

The formulae of this propositional temporal logic are constructed inductively in the normal way. Figure 2 summarises the definition. Thus, propositions p from $\mathcal{A}_P$ are formulae, then given formulae φ and ψ, so are the propositional combinations **true**, **false**, $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$ and $\varphi \Leftrightarrow \psi$, and the temporal combinations, $\bigcirc\varphi$, $\square\varphi$, $\lozenge\varphi$, $\bullet\varphi$, $\blacksquare\varphi$, $\blacklozenge\varphi$, $\varphi\mathcal{W}\psi$, $\varphi\mathcal{U}\psi$, $\varphi\ \mathcal{Z}\ \psi$ and $\varphi\mathcal{S}\ \psi$. The binding of the propositional connectives is assumed prioritized in the order given, i.e. (highest) $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ (lowest). The unary temporal connectives, i.e. $\bigcirc$, $\square$ and $\lozenge$, have binding equal to negation, i.e. $\neg$. The binary temporal connectives have binding in between $\neg$ and $\wedge$. However, as usual, to change the effect of binding, bracketing of formulae is allowed; so, given formula φ, then (φ) is also a formula.

The syntactic class of well-formed formulae defined in Figure 2 is referred to as *Wff*. It is also useful to define the subclasses $Wff_<$, $Wff_=$, $Wff_\geq$, covering, respectively, strict past time formulae, present time formula and non-strict future time formulae.

$Wff_=$ is the set which includes the propositions p from $\mathcal{A}_P$ and is closed under the propositional connectives.

$Wff_<$ is the set which includes the formulae $\bullet\phi$, $\blacksquare\phi$, $\blacklozenge\phi$, $\phi\mathcal{S}\ \psi$ and $\phi\ \mathcal{Z}\ \psi$, where ϕ and ψ are in $Wff_\leq$, and is closed under unary and binary propositional connectives. $Wff_\leq$ includes the set of propositions and is closed under the propositional and past-time temporal connectives.

$Wff_\geq$ is the set which includes $Wff_=$ and is closed under the propositional and future-time temporal connectives.

Proposition Symbols

$a, b \ldots \in \mathcal{A}_E$
$x, y \ldots \in \mathcal{A}_C$

Note: $\mathcal{A}_P = \mathcal{A}_E \cup \mathcal{A}_C$ and $\mathcal{A}_E \cap \mathcal{A}_C = \emptyset$

Classical Propositional Connectives

true **false**

$\neg$ $\wedge$ $\vee$ $\Rightarrow$ $\Leftrightarrow$

Temporal Connectives

unary	$\bigcirc$	next
	$\bullet$	last
	$\square$	always in future
	$\blacksquare$	always in past
	$\lozenge$	sometime in future
	⟐	sometime in past
binary	$\mathcal{W}$	unless
	$\mathcal{Z}$	zince
	$\mathcal{U}$	until
	$\mathcal{S}$	since

Figure 1: Basic Symbols

- A proposition p drawn from $\mathcal{A}_P$ is a formula.
- If φ and ψ are formulae, then so are
  - **true** **false**
  - $\neg\varphi$ $\varphi \wedge \psi$ $\varphi \vee \psi$ $\varphi \Rightarrow \psi$ $\varphi \Leftrightarrow \psi$
  - $\bigcirc\varphi$ $\square\varphi$ $\lozenge\varphi$
  - $\bullet\varphi$ $\blacksquare\varphi$ ⟐φ
  - $\varphi\mathcal{W}\psi$ $\varphi\mathcal{U}\psi$
  - $\varphi\,\mathcal{Z}\,\psi$ $\varphi\,\mathcal{S}\,\psi$
- If φ is a formula, then so is (φ).

Figure 2: Formula Definitions

2.1.2 Semantics

Model structures (σ, i) are used to provide an interpretation for the temporal formulae. Figure 3 summarises the structure. The σ component provides an interpretation for the atomic propositions of the language.

Model $\mathcal{M}$ is a structure (σ, i)

$$\sigma, i \;\in\; (\mathbf{N} \to 2^{\mathcal{A}_P}) \times \mathbf{N}$$

σ provides an interpretation for the propositions. Often, σ is treated as a *state* sequence

$$s_0 \quad s_1 \quad s_2 \quad s_3 \quad \ldots$$

Here, state s_j denotes a set of propositions given true by σ at j

i represents the current moment in time

Figure 3: Model Structure

Given some moment in time, represented by a natural number, j, $\sigma(j)$ is a set of propositions drawn from the alphabet $\mathcal{A}_P$ and denotes all those propositions that are to be taken true at that moment. σ is total with respect to $\mathbf{N}$. The i component of the model gives the index which is to represent the current moment, i.e. *now*, in the model (referred to as the *reference point* of the model). Clearly, indices greater than i represent future moments and indices less than i represent past moments.

The model is *linear* because for every moment j there is exactly one future, i.e. given by the points $j+1, j+2$, etc. The model is *discrete*: it is based on the natural numbers. Furthermore, as the mapping is assumed total over the naturals, time is infinite in the future and finite, or bounded, in the past. Because of the linearity and discreteness properties, the σ mapping can be conveniently represented by a sequence of *states*, e.g. $s_0, s_1, s_2, s_3, \ldots$.

A relation $\models$ is defined inductively over the structure of formulae and provides an interpretation for temporal formulae in the given model structures. Figure 4 provides its definition. Clearly, **true** is true in any model $\mathcal{M}$ and **false** is true in no model.

For propositions $p \in \mathcal{A}_P$, p is true in the model if and only if p is true in the current moment, i.e. is in the set $\sigma(i)$. The interpretations of the propositional connectives are defined in the obvious way.

The truth of a next time formula, e.g. $\bigcirc\varphi$, in the model (σ, i) is given by the truth of the formula φ in the successor model of (σ, i) , i.e. by the truth in the model $(\sigma, i+1)$.

The formula $\square\varphi$, read as "always φ", is true for a model (σ, i) if and only if the formula φ is true in all the future (including the current) models of (σ, i), i.e. the models $(\sigma, i+k)$ for every $k \in \mathbf{N}$. Thus, if $\square p$ is true for some model, then p will be true now and in every future moment of that model.

The diamond formula $\diamondsuit\varphi$, read as "eventually φ", is true in (σ, i) if and only if φ is true for the model σ, j with reference point, j, now or later than i $(j \geq i)$.

The strong until formula $\varphi\,\mathcal{U}\,\psi$ is true in model (σ, i) if and only if (i) ψ is true eventually in that model, say with reference point $i+k$, and (ii) the formula φ is true for all the models $(\sigma, i+j)$ where the reference points $i+j$ ranges up to but not including the reference point for which ψ is true. Note that the $\varphi\,\mathcal{U}\,\psi$ is true in any model for which ψ is true, i.e. $\psi \Rightarrow \varphi\,\mathcal{U}\,\psi$ is true for every model.

The past time temporal operators are defined to be strict past time versions of their future time counterparts, i.e. the past does not include the present moment. Since the model structure allows easy backwards as well as forwards reasoning the interpretations given should be self evident. However, it is worth discussing the beginning of time. What interpretation should be given to the $\bullet$ operator. We have given $\bullet$ a weak interpretation, i.e. for any formula φ, $\bullet\varphi$ is true at the beginning of time. Hence the $\bullet$false can be used to determine the beginning of time. One can define a strong (existential) last time operator $\circledcirc$ such

Propositions

$$\sigma, i \models p \quad \text{iff} \quad p \in \sigma(i)$$

Propositional Connectives

$$\begin{array}{lll} \sigma, i \models \mathbf{true} & & \\ \sigma, i \models \neg\varphi & \text{iff} & \text{not } \sigma, i \models \varphi \\ \sigma, i \models \varphi \wedge \psi & \text{iff} & \sigma, i \models \varphi \text{ and } \sigma, i \models \psi \\ \text{etc.} & & \end{array}$$

Temporal Connectives

$$\begin{array}{lll} \sigma, i \models \bigcirc\varphi & \text{iff} & \sigma, i+1 \models \varphi \\ \sigma, i \models \square\varphi & \text{iff} & \text{for } \mathbf{all}\ k \in \mathbf{N}\ \ \sigma, i+k \models \varphi \\ \sigma, i \models \Diamond\varphi & \text{iff} & \text{for } \mathbf{some}\ k \in \mathbf{N}\ \ \sigma, i+k \models \varphi \\ \sigma, i \models \varphi\,\mathcal{U}\,\psi & \text{iff} & \text{for } \mathbf{some}\ k \in \mathbf{N} \\ & & \quad \sigma, i+k \models \psi \text{ and} \\ & & \quad \text{for } \mathbf{all}\ j \in 0..k-1,\ \ \sigma, i+j \models \varphi \\ \sigma, i \models \varphi\,\mathcal{W}\,\psi & \text{iff} & \sigma, i \models \varphi\,\mathcal{U}\,\psi \text{ or } \sigma, i \models \square\varphi \\ \sigma, i \models \bullet\varphi & \text{iff} & \text{if } i > 0 \text{ then } \sigma, i-1 \models \varphi \\ \sigma, i \models \blacksquare\,\varphi & \text{iff} & \text{for } \mathbf{all}\ k \in 1..i\ \ \sigma, i-k \models \varphi \\ \sigma, i \models \Diamond\!\!\!\cdot\ \varphi & \text{iff} & \text{for } \mathbf{some}\ k \in 1..i\ \ \sigma, i-k \models \varphi \\ \sigma, i \models \varphi\,\mathcal{S}\,\psi & \text{iff} & \text{for } \mathbf{some}\ k \in 1..i \\ & & \quad \sigma, i-k \models \psi \text{ and} \\ & & \quad \text{for } \mathbf{all}\ j \in 1..k-1,\ \sigma, i-j \models \varphi \\ \sigma, i \models \varphi\,\mathcal{Z}\,\psi & \text{iff} & \sigma, i \models \varphi\,\mathcal{S}\,\psi \text{ or } \sigma, i \models \blacksquare\,\varphi \end{array}$$

Figure 4: Interpretation in Model Structure

that $\circledcirc\varphi$ is false at the beginning of time for any φ. Then, of course, notice the duality between the two last time operators, i.e. $\circledcirc\varphi \Leftrightarrow \neg \bullet \neg \varphi$.

Definition 2.1 *A formula φ is said to be satisfied by a model $\mathcal{M} = (\sigma, i)$ if and only if $\sigma, i \models \varphi$.*

(Alternatively, $\sigma, i \models \varphi$ can be read as φ is true in the model (σ, i).)
The definition of validity is as in classical logic.

Definition 2.2 *A formula φ is said to be valid, $\models \varphi$, if and only if it is true for all models $\mathcal{M}$ ($\mathcal{M} \models \varphi$).*

Theorem 2.1 *PML is decidable.*

Proof See [Gou84], for example. This is an important result. It means that algorithms can be constructed to determine the validity, or otherwise, of any given PML formula.

2.2 METATEM Programs and Execution

A METATEM program for controlling a reactive process is presented as a collection of temporal rules. The rules apply universally in time and determine how the process progresses from one moment to the next. A temporal rule is given in the following clausal form

past time antecedent $\Rightarrow$ future time consequent.

This is not an unnatural rule form and occurs in some guise in most programming languages. For example, in imperative programming languages it corresponds to the conditional statement. In declarative logic programming, we have the Horn clause rule form of Prolog and other similar languages. In METATEM, the "past time antecedent" is a temporal formula referring strictly to the past, i.e. it is in $Wff_{<}$; the "future time consequent" is a temporal formula referring to the present and future, i.e. it is in $Wff_{\geq}$. Although we can adopt a declarative interpretation of the rules, for programming and execution purposes we take an imperative reading following the natural way we ourselves tend to behave and operate, namely,

on the basis of the past **do** the future.

Given a program consisting of a set of rules R_i, this imperative reading results in the construction of a model for the formula $\Box \bigwedge_i R_i$, which we refer to as the program formula. The execution of a METATEM program proceeds, informally, in the following manner. Given some initial history of execution:

1. determine which rules currently apply, i.e. find those rules whose past time antecedents evaluate to true in the current history;

2. "jointly execute" the consequents of the applicable rules together with any commitments carried forward from previous times — this will result in the current state being completed and the construction of a set of commitments to be carried into the future;

3. repeat the execution process for the next moment in the context of the new commitments and the new history resulting from 2 above.

The "joint execution" of step 2 above relies upon a separation result for non-strict future-time temporal formulae, i.e. of the syntactic class $Wff_{\geq}$. Given a formula ψ in $Wff_{\geq}$, it can be written in a logically equivalent form as

$$\bigvee_{i=1}^{n} f_i$$

where each f_i is either of the form $\varphi_i \wedge \bigcirc \psi_i$ or of the form φ_i where each φ_i is a conjunction of literals[1], i.e. a present time formula from $Wff_{=}$, and ψ_i is a future (non-strict) time formula, i.e. in $Wff_{\geq}$. Thus φ_i

[1] A literal is either a proposition or its negation.

can be used to build the current state, and recursive application of the execute mechanism on ψ_i (with the program rules) will construct the future states.

It is worth mentioning that although the METATEM program rule form of "past implies future" is a syntactic restriction on *Wff*, it is not a semantic restriction. A Separation Theorem (see [Gab89] for proof details) establishes that arbitrary temporal formulae, of class *Wff*, can be written in the form

$$\Box(\bigwedge_{i=1}^{n}(\xi_i \Rightarrow \psi_i))$$

where ξ_i are strict past time formulae, i.e. in $Wff_{<}$, and ψ_i are non-strict future time formulae, i.e. in $Wff_{\geq}$[2].

2.3 Examples

To demonstrate the execution process we offer two simple examples. The first is a resource manager, the second is the ubiquitous "dining philosophers".

2.3.1 A Resource Manager

Consider a resource being shared between several (distributed) processes, for example a database lock. We require a "resource manager" that satisfies the constraints given in Figure 5.

1. If the resource is requested by a process then it must eventually be allocated to that process.
2. If the resource is not requested then it should not be allocated.
3. At any one time, the resource should be allocated to at most one process.

Figure 5: Resource Manager Constraints

To simplify the exposition of the execution process, we restrict the example to just two processes. Let us use propositions r_1 and r_2 to name the occurrence of a request for the resource from process 1 and process 2 respectively. Similarly, let propositions a_1 and a_2 name appropriate resource allocations. It is important to note that the difference between propositions r_i and a_i. The request propositions are those controlled by the environment of the resource manager, whereas the allocation propositions are under direct control of the resource manager and can not be effected by the environment. Writing the given informal specification in the desired rule form, i.e. "pure past formula implies present and future formula", results in the rules of Figure 6.

We take the "state" as consisting of the four propositions r_1, r_2, a_1 and a_2. Given particular settings for the environment propositions, i.e. r_1 and r_2, the execution will determine appropriate values for the allocation propositions. Figure 7 gives the first seven steps of a typical trace.

Step 0. The environment has requested the resource for process 1. To see how the current state is completed, we must find which rules from Figure 6 apply. Clearly rules 1 and 2 do not apply; we are currently at the beginning of time and hence $\circledcirc\,\phi$, for any ϕ, is false. The other rules do apply and require that both a_1 and a_2 are made false. No extra commitments are to be carried forward. Execution proceeds to the next time step.

[2] The future time formulae are restricted so that they contain no environment propositions.

$$\odot r_1 \Rightarrow \Diamond a_1 \tag{1}$$

$$\odot r_2 \Rightarrow \Diamond a_2 \tag{2}$$

$$\neg r_1 \; \mathcal{Z} \; (a_1 \wedge \neg r_1) \Rightarrow \neg a_1 \tag{3}$$

$$\neg r_2 \; \mathcal{Z} \; (a_2 \wedge \neg r_2) \Rightarrow \neg a_2 \tag{4}$$

$$\bullet\,\mathbf{true} \Rightarrow (\neg a_1 \vee \neg a_2) \tag{5}$$

Figure 6: Temporal Rules for Resource Manager

Requests		*Allocations*		*Time*	*Commitments*
r_1	r_2	a_1	a_2	*Step*	
y	n	n	n	0	
n	n	y	n	1	
y	y	n	n	2	
n	n	y	n	3	$\Diamond a_2$
y	n	n	y	4	
n	y	y	n	5	
n	n	n	y	6	

Figure 7: Sample Execution of Resource Manager

Step 1. First, there are no new requests from the environment. However, in examining which rules apply, we note that hypothesis of rule 1 is true (there was a request in the previous moment), hence we must "execute" $\Diamond a_1$. Also, rule 4 and rule 5 applies, in fact, the latter rule applies at every step because its hypothesis is always true. To execute $\Diamond a_1$, we execute $a_1 \vee (\neg a_1 \wedge \bigcirc \Diamond a_1)$. We have a choice; however, our mechanism will prioritise such disjunctions and attempt to execute them in a left to right order. One reason for this is that to satisfy an eventuality such as $\Diamond \phi$, we must eventually (in some future time step) satisfy ϕ; [3] so we try to satisfy it immediately. If we fail then we must carry forward the commitment to satisfy $\Diamond \phi$. Here we can make a_1 true. Rule 4 requires that a_2 is false, leaving rule 5 satisfied.

Step 2. The environment makes a request for both process 1 and process 2. The hypotheses of rules 1 and 2 are false, whereas those of rules 3 and 4 are true. So a_1 and a_2 are made false. No commitments are carried forward.

Step 3. No new requests from the environment, but there are two outstanding requests, i.e. both rule 1 and 2 apply. However, rule 5 requires that only one allocation may be made at any one time. The execution mechanism "chose" to postpone allocation to process 2. Thus, the eventuality from rule 1 is satisfied, but the eventuality for rule 2 is postponed, shown in the figure by carrying a commitment $\Diamond a_2$.

Step 4. A further request from the environment occurs, however, of interest here is the fact that $\Diamond a_2$ must be satisfied in addition to the commitments from the applicable rules. Note that this time, rule 4 does not apply as there is an outstanding request. Fortunately, the execution mechanism has no need to further postpone allocation to process 2 and a_2 is made true.

Steps 5 – 6 Similar to before.

2.3.2 Dining Philosophers

Our second example illustrates a rather simple solution to the dining philosophers problem. First, consider, if you can, a dining philosopher, Philosopher_A, alone in a closed environment. What is it that we wish to express about this philosopher? Basically that he never goes hungry, i.e. that he eats sufficiently often. So in our abstraction we use the proposition eaten_A to denote that Philosopher_A has just eaten. Hence a rather high level description of the behaviour of this chap might be given by the temporal logic formula

$$\Box \Diamond \text{eaten}_A.$$

However, since we know that he will eventually be rather more sociable and want to dine with colleagues, we'll give him a couple of forks and get him used not to using his fingers. In order to have eaten, we pretend that he needs to have possessed a fork in his left hand and one in his right hand for at least two moments; we denote possession of forks in left and right hands by the propositions fork_{AL} and fork_{AR}, respectively. Thus we should add to the above the formula

$$\Box(\text{eaten}_A \Rightarrow \circledcirc(\circledcirc(\text{fork}_{AL} \wedge \text{fork}_{AR}) \wedge (\text{fork}_{AL} \wedge \text{fork}_{AR}))).$$

Writing this as a METATEM process in rule form, we get the following

$$\bullet\, \textbf{true} \Rightarrow \Diamond \text{eaten}_A$$

$$\neg \circledcirc(\circledcirc(\text{fork}_{AL} \wedge \text{fork}_{AR}) \wedge (\text{fork}_{AL} \wedge \text{fork}_{AR})) \Rightarrow \neg \text{eaten}_A$$

which we will refer to as the Philosopher_A program. Let us execute, i.e. run, not kill, Philosopher_A. Remember that he is alone in his own closed world, with, therefore, everything under his own control. The interpretation process in fact will produce a run of Philosopher_A that has him eating infinitely often, that

[3] This particular choice of execution is described in more detail in section 3.1.2

is if it were given enough time. Although he doesn't necessarily eat continuously, one possible execution trace will satisfy the temporal formula $\Diamond \Box \text{eaten}_A$. For example, all moments have fork_{AL} and fork_{AR} true, but the first two moments have eaten_A false. The first few steps of a trace produced by a prototype METATEM interpreter [Fis88] are given in Figure 8. In the first step, the interpreter sets fork_{AL} true but fork_{AR} false. Since both forks have not been held by the philosopher for the last two moments, eaten_A must be made false and the satisfaction of $\Diamond \text{eaten}_A$ must be postponed. A commitment to satisfy $\Diamond \text{eaten}_A$ is carried forward to step 1. The birthdate of the commitment tags the formula. The next two steps have fork_{AL} and fork_{AR} set true, but again postponement of satisfaction of $\Diamond \text{eaten}_A$ has to occur. These new commitments are absorbed into the commitment to satisfy $\Diamond \text{eaten}_A$ from step 0. In step 3 the philosopher can eat, therefore no commitment is carried forward. In step 4, we have a repeat of the situation from step 0. It is more interesting to create a process with two, or more, philosophers. In such a situation,

Propositions			*Time*	*Commitments*
fork_{AL}	fork_{AR}	eaten_A	*Step*	
y	n	n	0	$\Diamond \text{eaten}_A(0)$
y	y	n	1	$\Diamond \text{eaten}_A(0)$
y	y	n	2	$\Diamond \text{eaten}_A(0)$
n	n	y	3	
y	n	n	4	$\Diamond \text{eaten}_A(4)$
y	y	n	5	$\Diamond \text{eaten}_A(4)$
y	y	n	6	$\Diamond \text{eaten}_A(4)$
n	n	y	7	
y	n	n	8	$\Diamond \text{eaten}_A(8)$

Figure 8: Sample Trace of Single Philosopher

the philosophers need to share forks, i.e. there should only be two real forks for two philosophers, three forks for three philosophers, etc. Consider just two philosophers. Assuming fork_{BL} and fork_{BR} are the propositions indicating that Philosopher_B has left and right forks, the following interfacing constraint needs to be placed.

$$\Box(\neg(\text{fork}_{AL} \wedge \text{fork}_{BR}) \wedge \neg(\text{fork}_{AR} \wedge \text{fork}_{BL}))$$

In rule form we thus have:

$$\begin{aligned}
\bullet\,\mathbf{true} &\Rightarrow \Diamond \text{eaten}_A \\
\neg \circledcirc(\circledcirc(\text{fork}_{AL} \wedge \text{fork}_{AR}) \wedge (\text{fork}_{AL} \wedge \text{fork}_{AR})) &\Rightarrow \neg \text{eaten}_A \\
\bullet\,\mathbf{true} &\Rightarrow \Diamond \text{eaten}_B \\
\neg \circledcirc(\circledcirc(\text{fork}_{BL} \wedge \text{fork}_{BR}) \wedge (\text{fork}_{BL} \wedge \text{fork}_{BR})) &\Rightarrow \neg \text{eaten}_B \\
\bullet\,\mathbf{true} &\Rightarrow \neg(\text{fork}_{AL} \wedge \text{fork}_{BR}) \\
\bullet\,\mathbf{true} &\Rightarrow \neg(\text{fork}_{AR} \wedge \text{fork}_{BL})
\end{aligned}$$

If we now consider execution of this set of rules, we should observe that both philosophers do eventually eat. Imagine the following potentially disastrous execution, e.g. Figure 9, from Philosopher_A's point of view. The interpreter decides to allocate both forks to Philosopher_B forever, i.e. it attempts to maintain $\neg\text{fork}_{AL}$, $\neg\text{fork}_{AR}$, fork_{BL} and fork_{BR} true. Thus the interpreter is not forced to make eaten_B false and hence will always be able to satisfy the constraint $\Diamond \text{eaten}_B$. However, the interpreter is forced to make eaten_A false,

Propositions						*Time*	*Commitments*
fork_{AL}	fork_{AR}	eaten_A	fork_{BL}	fork_{BR}	eaten_B	*Step*	
n	n	n	y	y	n	0	$\Diamond\text{eaten}_A(0)$, $\Diamond\text{eaten}_B(0)$
n	n	n	y	y	n	1	$\Diamond\text{eaten}_A(0)$, $\Diamond\text{eaten}_B(0)$
n	n	n	y	y	y	2	$\Diamond\text{eaten}_A(0)$
n	n	n	y	y	y	3	$\Diamond\text{eaten}_A(0)$
n	n	n	y	y	y	4	$\Diamond\text{eaten}_A(0)$
n	n	n	y	y	y	5	$\Diamond\text{eaten}_A(0)$
n	n	n	y	y	y	6	$\Diamond\text{eaten}_A(0)$

Figure 9: Bad Trace for Two Philosophers

because the appropriate fork propositions are not true, and hence the satisfaction of $\Diamond\text{eaten}_A$ is continually postponed. Eventually the loop checking mechanism of the interpreter recognises this potential continual postponement of $\Diamond\text{eaten}_A$ since step 0 and backtracks to make a different choice. This loop checking mechanism will eventually cause both forks to be "allocated" to Philosopher_A. So the disaster, in fact, does not occur and both philosophers eat infinitely often.

Figure 10 gives part of a trace that can be obtained with the METATEM interpreter for this example.

Propositions						*Time*	*Commitments*
fork_{AL}	fork_{AR}	eaten_A	fork_{BL}	fork_{BR}	eaten_B	*Step*	
y	n	n	n	n	n	0	$\Diamond\text{eaten}_A(0)$, $\Diamond\text{eaten}_B(0)$
y	y	n	n	n	n	1	$\Diamond\text{eaten}_A(0)$, $\Diamond\text{eaten}_B(0)$
y	y	n	n	n	n	2	$\Diamond\text{eaten}_A(0)$, $\Diamond\text{eaten}_B(0)$
n	n	y	y	y	n	3	$\Diamond\text{eaten}_B(0)$
n	n	n	y	y	n	4	$\Diamond\text{eaten}_A(4)$, $\Diamond\text{eaten}_B(0)$
y	y	n	n	n	y	5	$\Diamond\text{eaten}_A(4)$
y	y	n	n	n	n	6	$\Diamond\text{eaten}_A(4)$, $\Diamond\text{eaten}_B(6)$
n	n	y	y	y	n	7	$\Diamond\text{eaten}_B(6)$
n	n	n	y	y	n	8	$\Diamond\text{eaten}_A(8)$, $\Diamond\text{eaten}_B(6)$

Figure 10: Sample Trace of Two Philosophers

2.4 An Interpreter for Propositional METATEM

Figure 11 contains, in a Prolog-like notation, a basic interpreter for propositional METATEM. A predicate **execute** is defined which takes three arguments: **rules**, a list of the program rules; **commitments**, a list of commitments that need to be satisfied; and **history**, a sequence of states to be regarded as the history so far.

Below we give a brief description of each of the predicates used in the interpreter. A more detailed description of aspects relating to the predicates **choose** and **build-new-state** is given in Section 3.

```
execute([],[],history) :- true.

execute(rules,commitments,history) :-
        check-past(rules, history, constraints),
        rewrite(constraints, commitments, exec-form),
        choose(exec-form, rules, current-constraints, new-commitments),
        build-new-state(history, current-constraints, new-history),
        execute(rules, new-commitments, new-history).
```

Figure 11: A basic interpreter for METATEM.

- **check-past(rules, history, constraints)**
  This predicate, when satisfied, has the effect of first determining the activated rules from **rules**, i.e. those rules that have past-time antecedents satisfied on **history**. The variable **constraints** is then bound to a list of the future parts of the activated rules. Thus, **constraints** contains all new commitments to be satisfied in any future execution.

- **rewrite(constraints, commitments, exec-form)**
  The rewrite predicate has the effect of rewriting the conjunction of all the formulae in **constraints** and those from **commitments** as a formula in standard form. This formula is bound to **exec-form** and represents the formula to be executed in the current state.

  The standard form adopted is one of

  $$\textbf{true}, \quad \text{or} \quad \textbf{false}, \quad \text{or} \quad \bigvee_{i=1}^{n\geq 1} f_i$$

  where each f_i is either of the form

  $$\bigwedge_{j=1}^{m_i\geq 0} \lambda_{ij} \wedge \bigcirc \varphi_i$$

  or

  $$\bigwedge_{j=1}^{m_i\geq 1} \lambda_{ij}$$

  and where λ_{ij} is a literal and $\varphi_i \in Wff_{\geq}$.

  Note that all the future time temporal connectives for PML can be translated into disjunctions of present and next-time formulae. For example, the following rewrite rules are sound

  $$\begin{aligned} a\mathcal{U}b &\longrightarrow b \vee (a \wedge \neg b \wedge \bigcirc a\mathcal{U}b) \\ a\mathcal{W}b &\longrightarrow b \vee (a \wedge \neg b \wedge \bigcirc a\mathcal{W}b) \\ \square a &\longrightarrow a \wedge \bigcirc \square a \\ \lozenge a &\longrightarrow a \vee (\neg a \wedge \bigcirc \lozenge a) \end{aligned}$$

  Furthermore, we use rewrite rules such as

  $$\begin{aligned} \bigcirc a \wedge \bigcirc b &\longrightarrow \bigcirc(a \wedge b) \\ \bigcirc a \vee \bigcirc b &\longrightarrow \bigcirc(a \vee b) \end{aligned}$$

  to ensure that each disjunct of the standard form contains at most one next-time formula.

As an example, consider the resource controller described above, where the rules (2), (3), and (5) are the ones whose past-time parts are satisfied on the current history. The list bound to **constraints** is, therefore,

$$[\Diamond a_2,\ \neg a_1,\ \neg a_1 \vee \neg a_2]$$

Now, assuming that the **commitments** carried over from the previous state is the list

$$[\Diamond a_1]$$

then the formulae from both lists are conjoined to form

$$\Diamond a_2 \ \wedge\ \neg a_1 \ \wedge\ (\neg a_1 \vee \neg a_2) \ \wedge\ \Diamond a_1$$

and then rewritten in the standard form as

$$(\neg a_1 \ \wedge\ a_2 \ \wedge\ \bigcirc \Diamond a_1) \ \vee\ (\neg a_1 \ \wedge\ \neg a_2 \ \wedge\ \bigcirc(\Diamond a_2 \ \wedge\ \Diamond a_1)).$$

This resulting formula is bound to **exec-form**.

- **choose(exec-form, rules, current-constraints, new-commitments)**
  The predicate **choose** has the effect of picking a disjunct to execute from **exec-form** and binding propositional constraints to **current-constraints** and next-state commitments to **new-commitments** (with the principal $\bigcirc$ stripped off). Clearly, several different strategies could be adopted for picking a disjunct, some of which are described in Section 3. Continuing the example above, if we assume that the disjunct containing the fewest temporal subformulae is chosen, this gives

  $$\neg a_1 \ \wedge\ a_2 \ \wedge\ \bigcirc \Diamond a_1$$

  current-constraints is the list $[\neg a_1,\ a_2]$ together with bindings for any other unbound propositions[4], and **new-commitments** is

  $$[\Diamond a_1]$$

  So, if the only component propositions are a_1 and a_2, **current-constraints** is simply

  $$[\neg a_1,\ a_2].$$

  This gives the propositional interpretation for the new state.

- **build-new-state(history, current-constraints, new-history)**
  The effect of this predicate is to build a new state by binding propositions in accordance with **current-constraints**. A loop-checking mechanism is invoked to ensure that the state about to be constructed does not perpetuate a situation where eventualities (such as $\Diamond \varphi$) are continually postponed[5]. If such postponement is detected, **build-new-state** fails and backtracking to a previous choice point occurs.

  If backtracking does not occur, the values of appropriate environment propositions are recorded and the new state generated is then appended to **history** to form **new-history**.

3 METATEM Interpreter Strategies

The predicate **choose** is the heart of the interpretation process. Its effect is to take the currently executable formula and the current program rules, and to choose a disjunct from the formula to execute, constructing the current constraints and the future continuation for that execution. The choices for this predicate are categorised as follows:

[4]Though other strategies can be used, e.g. see Section 3, we will assume for the moment that random values in $\{true, false\}$ are given to these other propositions.

[5]This *loop-checking* mechanism is briefly described in Section 3.3.

- Choosing a disjunct to execute.
  Given the executable formula in standard form, the disjunct to be executed can be chosen in several ways; for example,

  1. by non-deterministic choice,
  2. by using an ordering on the disjuncts, or,
  3. by choosing a disjunct that is most likely to lead to a successful execution.

- Binding strategies for variables.
  Once a particular disjunct has been chosen for execution, the bindings for the propositions constrained by that disjunct are determined. However, the choice of bindings for other propositions remains. The choice for these bindings can be made

  1. non-deterministically
  2. by choosing a binding that is most likely to lead to a successful execution, or,
  3. by delaying bindings until they are needed.

The following two subsections described these choice mechanisms in more detail.

3.1 Choosing a disjunct

3.1.1 Non-deterministic choice of disjunct

Making a non-deterministic choice of disjunct to execute can be problematic; the interpreter could possibly delay the satisfaction of eventualities indefinitely. Consider, for example, the execution of $\Diamond p$. This can be rewritten as

$$p \vee (\neg p \wedge \bigcirc \Diamond p)$$

If we make a non-deterministic choice then it is possible to always choose the second disjunct, i.e. $\neg p \wedge \bigcirc \Diamond p$, making p false in every state, thus delaying indefinitely the time when p becomes true. Although a loop-checking mechanism, see Section 3.3, would eventually detect such a situation, it is desirable that excessive backtracking be avoided by use of an alternative choice mechanism.

3.1.2 Selection by order

The disjuncts of the executable formula can be ordered, then the predicate **choose** may select disjuncts following that ordering. There are several ways to obtain such an ordering, two of which are described below.

- Disjuncts are ordered by the number of eventualities which they contain, and then by the number of propositions which they leave unbound. However, consider the execution of the following set of rules:

$$\begin{array}{rcl} \newcommand{\start}{\bullet} \bullet\,\mathbf{false} & \Rightarrow & \Diamond a \\ \bullet\,\mathbf{true} & \Rightarrow & \Diamond b \wedge \Diamond c \\ \bullet\,\mathbf{true} & \Rightarrow & (\neg a \wedge b \wedge c) \vee (\neg b \wedge \neg c) \end{array}$$

  which expands, in the first state, to

$$\begin{array}{l} (\neg a \wedge b \wedge c \wedge \bigcirc \Diamond a) \quad \vee \\ (a \wedge \neg b \wedge \neg c \wedge \bigcirc (\Diamond b \wedge \Diamond c)) \quad \vee \\ (\neg a \wedge \neg b \wedge \neg c \wedge \bigcirc (\Diamond a \wedge \Diamond b \wedge \Diamond c)) \end{array}$$

The ordering described would thus cause the left-most disjunct to be selected postponing the satisfaction of $\Diamond a$. However, in all subsequent states, the above would then remain the executable formula. In this case, the execution of a would again be delayed indefinitely.

- To avoid such possibilities, we introduce a further ordering on disjuncts to be applied after the above ordering has been carried out and then only on disjuncts containing postponed eventualities. This further ordering orders the disjuncts so that those that contain the oldest eventualities appear last in the ordering. For example, using this strategy on the above formula, the executable formula in the first state could be, as above,

$$\begin{array}{l}(\neg a \wedge b \wedge c \wedge \bigcirc \Diamond a) \quad \vee \\ (a \wedge \neg b \wedge \neg c \wedge \bigcirc(\Diamond b \wedge \Diamond c)) \quad \vee \\ (\neg a \wedge \neg b \wedge \neg c \wedge \bigcirc(\Diamond a \wedge \Diamond b \wedge \Diamond c))\end{array}$$

If the leftmost disjunct were chosen, then in the next state, because the $\Diamond a$ eventuality remained unsatisfied, the executable formula would be

$$\begin{array}{ll}(a \wedge \neg b \wedge \neg c \wedge \bigcirc(\Diamond b \wedge \Diamond c)) \quad \vee & (1) \\ (\neg a \wedge \neg b \wedge \neg c \wedge \bigcirc(\Diamond a \wedge \Diamond b \wedge \Diamond c)) \quad \vee & (2) \\ (\neg a \wedge b \wedge c \wedge \bigcirc \Diamond a) & (3)\end{array}$$

and the a would be executed. Note that the eventuality $\Diamond a$ has been present since the previous state, state 1, while both $\Diamond b$ and $\Diamond c$ were generated in state 2. Thus the oldest postponed eventuality in disjunct 1 is from state 2, while the oldest postponed eventualities in disjuncts 2 and 3 are from state 1.

The executable formula for the third state would then be

$$(\neg a \wedge b \wedge c) \vee (\neg b \wedge \neg c \wedge \bigcirc(\Diamond b \wedge \Diamond c))$$

and choosing the left-most disjunct satisfies the remaining eventualities immediately.

Thus, by always ordering the disjuncts such that all satisfying subformulae for the longest outstanding eventuality occur in the leftmost disjuncts, or equivalently that the oldest postponed eventuality occurs in the rightmost disjunct, we are guaranteed to avoid the delay of eventualities described above.

Though this strategy is inefficient in particular cases, it is simple and forms a useful starting point from which to develop more sophisticated heuristics.

3.1.3 Choice directed by future satisfiability

An alternative approach is to attempt to eliminate disjuncts that can not lead to a successful execution. In the case of PML, the satisfiability of a chosen disjunct, which is a future-time formula, can be determined. If no disjunct is satisfiable, then there is no possible execution path from this state; otherwise, there may be some execution path that can be followed[6].

Suppose the following program is to be executed

$$\CIRCLE\, \textbf{false} \Rightarrow \Diamond a \wedge \bigcirc \Box \neg a \wedge \Diamond \neg a.$$

In the first state, the executable formula is

$$(\neg a \wedge \bigcirc(\Diamond a \wedge \Box \neg a)) \vee (a \wedge \bigcirc(\Diamond \neg a \wedge \Box \neg a))$$

[6]Note that when the formula is satisfiable, the model constructed can be used to help the interpreter find an executable path.

The previous choice strategy would result in an attempt to execute the first disjunct. However, a satisfiability check would eliminate this disjunct from consideration.

There are two major problems with this approach. The first is that when using more general versions of METATEM (e.g. first order) satisfiability is not decidable. The other is that satisfiability in PML is PSPACE-complete, which has obvious implications for the speed of the execution process.

3.2 Binding strategies

When a disjunct has been chosen for execution, it may not fully determine the values of all propositions in the current state, in which case, bindings for the unconstrained propositions must be found.

As in Section 3.1, the choice of binding can be made using non-deterministic or satisfiability directed strategies. Unfortunately, these binding strategies suffer the same problems as their choice strategy counterparts, for example excessive backtracking. One way to avoid some of these problems is to delay the binding of propositions until their values are neeeded.

3.2.1 Delayed binding

Consider the following set of rules:

$$\begin{aligned} \circledcirc\circledcirc\circledcirc p &\Rightarrow \Box\neg a_1 \\ \circledcirc r_1 &\Rightarrow \Diamond a_1 \end{aligned}$$

If all propositions are bound at each computation step then an arbitrary value is chosen for p. Suppose p is set to true in the first state and that the environment keeps r_1 false until the third state. In the fourth state, the premise of the first rule is true and henceforth a_1 must be false. However, the interpreter also has to satisfy $\Diamond a_1$. After several more steps the fact that an unsatisfiable loop has been generated is detected and execution fails causing a backtrack to state 1 where p was set. p is then set to false in the first state and when the r_1 occurs again, it can be satisfied[7].

By delaying the binding of p in the first state until it is needed, much of this backtracking can be avoided. The binding of p can be delayed until the fourth state when the value of p in the first state is required. In the fourth state, the commitments can be examined and a suitable value for p in state 1 chosen; in the above scenario p would be retrospectively set to false in state 1.

One problem with this approach is that propositions may be unbound for an indeterminate length of time. However, the values of such propositions may be required by the environment.

3.3 Loop-checking

The disjunctive nature of formulae involving eventualities, e.g., $\Diamond\varphi$, $\varphi\,\mathcal{U}\,\psi$, will often lead to postponement of their satisfaction. If the formula being executed contains an unsatisfiable eventuality then this continual postponement can be seen as a loop. The problem that then arises is how to detect such loops and then to distinguish between genuine loops that are due to unsatisfiable eventualities and apparent loops that have satisfiable extensions.

For example, if the formula to be executed were

$$\Diamond a \wedge \Box\neg a$$

which expands to

$$\neg a \wedge \bigcirc(\Diamond a \wedge \Box\neg a)$$

[7]Note that the satisfiability directed choice of binding described above would not help the choice of the binding for p as no matter what p is bound to initially, in state 4 the formula to be executed is still satisfiable. In this case it is only when an r_1 occurs that satisfiability is potentially lost.

then the eventuality $\Diamond a$ would be continuously postponed, as the formula is not satisfiable. In the example below, the eventuality $\Diamond q$ is postponed but only for a finite time. Indeed, it can only be satisfied int the fifth state when q can be set to true.

$$\begin{array}{rcl} \bullet\bullet\bullet\bullet p & \Rightarrow & \neg q \\ \bullet\,\mathbf{false} & \Rightarrow & \neg p \wedge \bigcirc \Box p \wedge \Diamond q \end{array}$$

Using the above execution mechanism there is a bound, within which any satisfiable eventuality must be satisfied. This bound, which depends on the size of the formula, enables us to distinguish between real and apparent loops. Thus the loop-checking mechanism causes the execution to fail and backtrack if an eventuality has been outstanding for more than the fixed number of states.

A coarse bound can be given by the number of different states that appear in the semantic tableau for the program formula, f. This bound is $2^{5|f|}$ where $|f|$ is the length of the formula f [LP85]. (Note that such a bound can not be established where arbitrary environment interaction occurs.) It can be shown that the execution mechanism described in Section 3.4 satisfies any satisfiable eventuality within such a fixed number of states.

3.4 Propositional METATEM Interpreter Correctness

Using the choice strategy based on ordering by eventuality age (described in Section 3.1.2) and the loop-checking mechanism described above defines an execution function, *execute*, for PML without environment propositions. This function takes a METATEM program and returns the model constructed during execution (unless the execution fails in which case an empty model is returned). We can then establish the following:

Theorem 3.1 *Let P be a program consisting of a set of program rules $\{R_i \mid i \in I\}$, each of the form $\xi_i \Rightarrow \psi_i$ where $\xi_i \in Wff_<$ and $\psi_i \in Wff_\geq$. If the program formula, f, defined as $\Box \bigwedge_{i\in I} R_i$ is satisfiable then execute($\bigwedge_{i\in I} R_i$,* **true**, []) *will generate a model for f.*

Theorem 3.2 *Let P be a program consisting of a set of program rules $\{R_i \mid i \in I\}$, each of the form $\xi_i \Rightarrow \psi_i$ where $\xi_i \in Wff_<$ and $\psi_i \in Wff_\geq$. If execute($\bigwedge_{i\in I} R_i$,* **true**, []) *does not fail, then the program formula f, defined as $\Box \bigwedge_{i\in I} R_i$ is satisfiable and execute generates a model for f.*

Outline proofs of these results are given in Appendix A.

4 Towards a First-order METATEM

In this section we consider a subset of first-order temporal logic, called FML, that can be executed in a similar manner to PML. Program rules are again written in the 'past implies future' form

$$\xi(\bar{x}) \;\Rightarrow\; \psi(\bar{x}, \bar{y})$$

where $\bar{x}$ and $\bar{y}$ represent tuples of variables, implicitly quantified as follows.

$$\forall \bar{x}.\ \exists \bar{y}.\ \xi(\bar{x}) \;\Rightarrow\; \psi(\bar{x}, \bar{y})$$

In FML the set of predicates is partitioned into component and environment predicates and, as in propositional METATEM, environment literals are not allowed in the future-time part of FML clauses.

Before describing an execution mechanism for FML, we will outline the syntax and semantics of FML.

4.1 Syntax and Semantics of FML

4.1.1 Syntax

The language FML consists of the following symbols

- A countable set, $\mathcal{A}_P$, of *predicate symbols*, each with an associated non-negative integer, its *arity*; predicate symbols of arity 0 are called *proposition symbols*. The set $\mathcal{A}_P$ is the disjoint union of the sets $\mathcal{A}_{PC}$ and $\mathcal{A}_{PE}$, of *component* and *environment* predicate symbols.
- A countable set, $\mathcal{A}_F$, of *function symbols*, each with an associated arity; function symbols of arity 0 are called *constant symbols*.
- A infix binary predicate symbol =.
- A countable set, $\mathcal{A}_V$, of *variable symbols*. The set $\mathcal{A}_V$ consists of the disjoint union of the sets $\mathcal{A}_{V\exists}$ and $\mathcal{A}_{V\forall}$, of *existential* and *universal* variables.
- Quantifiers $\exists$, $\forall$.
- A set of propositional and temporal connectives, as in PML.

A *term* is defined by

- a variable symbol is a term,
- if f is a function symbol of arity n, and $t_1, t_2, \ldots t_n$ are terms then $f(t_1, t_2, \ldots t_n)$ is a term.

Denote by $\mathcal{G}$ the set of *ground terms*, i.e. terms containing no occurrences of variables. A *binding* is then a partial function $\alpha : \mathcal{A}_V \to \mathcal{G}$.
A function η which applies bindings, α, to terms is now defined.

- If $x \in \mathcal{A}_V$ is in the domain of α, then $\eta(\alpha, x) = \alpha(x)$.
- If $x \in \mathcal{A}_V$ is not in the domain of α, then $\eta(\alpha, x) = x$.
- If f is an n-ary function symbol and $t_1, t_2, \ldots t_n$ are terms, then

$$\eta(\alpha, f(t_1, t_2, \ldots t_n)) = f(\eta(\alpha, t_1), \eta(\alpha, t_2), \ldots \eta(\alpha, t_n)).$$

An *atomic formula* is defined to be an application of a predicate to term arguments, i.e. if p is a predicate symbol of arity n, and $t_1, t_2, \ldots t_n$ are terms then $p(t_1, t_2, \ldots t_n)$ is an atomic formula A *grounded atomic formula* is an atomic formula containing no occurrences of variables.
We also assume the existence of a countable set, $\mathcal{A}_T$, of atoms, which is a subset of the set of all grounded atomic formulae. The existence of this set enables us to restrict the domains of individual predicate and function symbols and also to implicitly restrict quantification; for example, if p is a predicate symbol then, in the expression $\forall x.p(x)$, x ranges only over those ground terms t for which $p(t) \in \mathcal{A}_T$.
The *well formed formulae* of the language are now defined by

- If φ is an atomic formula such that, for some binding α, $\eta(\alpha, \varphi) \in \mathcal{A}_T$, then φ is a well-formed formula.
- If t_1, t_2 are terms, then $t_1 = t_2$ is a well-formed formula.
- Well-formed formulae are constructed from other formulae by application of propositional and temporal connectives as in PML.
- If $x \in \mathcal{A}_{V\forall}$ and φ is a well-formed formula, then $\forall x.\varphi$ is a well-formed formulae.
- If $y \in \mathcal{A}_{V\exists}$ and φ is a well-formed formula, then $\exists y.\varphi$ is a well-formed formulae.

The definition of the function η is extended in the obvious way over well-formed formulae.

4.1.2 Semantics

Well-formed formulae of FML are interpreted over model structures of the form (σ, α, i) where

$$\sigma : \mathbf{N} \to 2^{\mathcal{A}_T}, \quad \alpha \text{ is a binding, and } i \in \mathbf{N}$$

The satisfaction relation $\models$ of PML is extended over model structures and well-formed formulae of FML as follows

$(\sigma, \alpha, i) \models \varphi$	only if $\eta(\alpha, \varphi)$ is a well-formed formula containing no free variables
$(\sigma, \alpha, i) \models p(t_1, t_2, \ldots t_n)$	iff $p(\eta(\alpha, t_1), \eta(\alpha, t_2), \ldots \eta(\alpha, t_n)) \in \sigma(i)$
$(\sigma, \alpha, i) \models t_1 = t_2$	iff $\eta(\alpha, t_1) = \eta(\alpha, t_2)$
$(\sigma, \alpha, i) \models \forall x.\varphi$	iff for *all* $\gamma \in \mathcal{G}$, such that $\eta(\alpha \dagger [x \mapsto \gamma], \varphi)$ is a well-formed formula[8], $(\sigma, \alpha \dagger [x \mapsto \gamma], i) \models \varphi$
$(\sigma, \alpha, i) \models \exists y.\varphi$	iff for *some* $\gamma \in \mathcal{G}$, such that $\eta(\alpha \dagger [y \mapsto \gamma], \varphi)$ is a well-formed formula, $(\sigma, \alpha \dagger [y \mapsto \gamma], i) \models \varphi$

Note that the above semantics imply a *rigid* interpretation of variables, i.e. the binding of a variable name is fixed over time.

4.2 Executing First-Order METATEM

The execution mechanism for FML proceeds in a similar manner to that of PML, see Figure 12 for an outline interpreter in Prolog-like notation. As with the execution of PML, a predicate **execute** is defined which takes, as arguments, a list of the program rules, a list of commitments that need to be satisfied, and a sequence of states to be regarded as the history so far. We use in our specification rules of the form

$$\xi(x_1, \ldots, x_m) \Rightarrow \psi(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

where the variables of ξ appear in ψ. These variables are interpreted universally and the extra variables y_i are interpreted existentially. ξ is a past formula, and ψ is a present or future formula. Note that this is in contrast to the Logic Programming convention where the additional variables in the antecedent are read existentially. Thus when we query the past with $\xi(x_1, \ldots, x_m)$ as a query, we must find all instantiations t_i of x_i for which $\xi(x_1, \ldots, x_m)$ is true and execute $\psi(t_0, \ldots, t_m, y)$ by finding at least one $f_j(t_1, \ldots, t_m)$, for each j, for which $\psi(t_1, \ldots, t_m, f_1(t_1, \ldots, t_m), \ldots, f_n(t_1, \ldots, t_m))$ is true.

Before outlining the predicates used in the FML interpreter, we give its formal derivation.

4.2.1 Derivation of Execution Mechanism for FML

Assume the following is given in the context of the above definition of FML.

1. A set of program rules, R_i, $i \in I$, each of the form

$$\forall \overline{x_i}.\ \exists \overline{y_i}.\ \xi_i(\overline{x_i}) \Rightarrow \psi_i(\overline{x_i}, \overline{y_i})$$

   where

$\overline{x_i}$	is a tuple of distinct universal variables,
$\overline{y_i}$	is a tuple of distinct existential variables,
$\xi_i(\overline{x_i})$	is a pure past time formula dependent on $\overline{x_i}$,

[8]Here $\alpha \dagger [x \mapsto \gamma]$ denotes the binding obtained by overwriting α with the binding $[x \mapsto \gamma]$.

$\psi_i(\overline{x_i}, \overline{y_i})$ is a future time formula dependent on $\overline{x_i}$ and $\overline{y_i}$.

2. A future time formula C, known as the commitment formula.
3. A finite prefix, or history sequence of states, σ_h, each state being a subset of the atoms defining FML.
4. An infinite state sequence continuation, σ.

Let $\mathcal{M}(\alpha)$ denote the model structure $(\sigma_h\hat{}\sigma, \alpha, |\sigma_h|)$. We now derive conditions that establish

$$\mathcal{M}([]) \models \Box \bigwedge_{i \in I} R_i \wedge C \tag{6}$$

First note that we can expand $\Box$ in 6 to give

$$\mathcal{M}([]) \models \bigwedge_{i \in I} \forall \overline{x_i}\, \exists \overline{y_i}\, \xi_i(\overline{x_i}) \Rightarrow \psi_i(\overline{x_i}, \overline{y_i}) \wedge C \wedge \bigcirc \Box \bigwedge_{i \in I} R_i \tag{7}$$

By appropriate renaming, the variables in each tuple $\overline{x_i}$ can be made unique. Without any loss in generality we assume this is the case, i.e. letting $vars = \{x \mid x \in \textbf{elems}\ \overline{x_i}, i \in I\}$, $|vars| = \Sigma_{i \in I} \textbf{len}\ \overline{x_i}$. The universal quantifiers in 7 can be thus be moved outwards over the conjunction. Independently, the existential quantifiers can be moved inwards; each $\xi_i(\overline{x_i})$ formula is independent of $\overline{y_i}$. The following, therefore, needs to be satisfied.

$$\mathcal{M}([]) \models \forall \overline{x_i}_{,i \in I}.\ \bigwedge_{i \in I} \xi_i(\overline{x_i}) \Rightarrow \exists \overline{y_i}.\ \psi_i(\overline{x_i}, \overline{y_i}) \wedge C \wedge \bigcirc \Box \bigwedge_{i \in I} R_i \tag{8}$$

By the definition of the semantics of universal quantification, for 8 to hold, the formula

$$\bigwedge_{i \in I} \xi_i(\overline{x_i}) \Rightarrow \exists \overline{y_i}.\ \psi_i(\overline{x_i}, \overline{y_i})$$

must be true for all bindings giving assignments to the variables in the tuples $\overline{x_i}$. As $\xi_i(\overline{x_i})$ is a pure past time formula, bindings for $\overline{x_i}$ can be determined from the history sequence σ_h.
Define J to be largest subset of I and B the largest set of bindings α each with domain $\bigcup_{j \in J} \textbf{elems}\ \overline{x_j}$ such that for all state sequence extensions σ

$$\text{for all } \alpha \in B,\quad \mathcal{M}(\alpha) \models \bigwedge_{j \in J} \xi_j(\overline{x_j}) \tag{9}$$

In effect, the set J defines the set of rules which are activated in this state and B gives the set of bindings that apply. Thus to satisfy 8 we need now just satisfy 10 below.

$$\text{for all } \alpha \in B,\quad \mathcal{M}(\alpha) \models \bigwedge_{j \in J} \exists \overline{y_j}.\ \psi_j(\overline{x_j}, \overline{y_j}) \wedge C \wedge \bigcirc \Box \bigwedge_{i \in I} R_i \tag{10}$$

At this stage, we apply the substitution function η, defined in Section 4.1.2 to remove the universal variable references. That is to say, 10 is semantically equivalent to 11 below.

$$\mathcal{M}([]) \models \bigwedge_{\alpha \in B} \eta(\alpha, \bigwedge_{j \in J} \exists \overline{y_j}.\ \psi_j(\overline{x_j}, \overline{y_j}) \wedge C \wedge \bigcirc \Box \bigwedge_{i \in I} R_i) \tag{11}$$

Since $C \wedge \bigcirc \Box \bigwedge_{i \in I} R_i$ have no free variables, the formula can be moved out of scope of η; 11 leads to 12.

$$\mathcal{M}([]) \models \bigwedge_{\alpha \in B} \eta(\alpha, \bigwedge_{j \in J} \exists \overline{y_j}.\ \psi_j(\overline{x_j}, \overline{y_j})) \wedge C \wedge \bigcirc \Box \bigwedge_{i \in I} R_i \tag{12}$$

By definition, η distributes over conjunctions, thus 13 is obtained.

$$\mathcal{M}([]) \models \bigwedge_{j \in J} \bigwedge_{\alpha \in B} \eta(\alpha, \exists \overline{y_j}.\ \psi_j(\overline{x_j}, \overline{y_j})) \wedge C \wedge \bigcirc \Box \bigwedge_{i \in I} R_i \tag{13}$$

For each distinct $\alpha \in B$ choose a distinct k from K and let $G_{jk}(\overline{y_j})$ denote the formula resulting from the substitution given by $\eta(\alpha, \psi_j(\overline{x_j}, \overline{y_j}))$. This leads to 14.

$$\mathcal{M}([]) \models \bigwedge_{j\in J} \bigwedge_{k\in K} \exists\overline{y_j}.\ G_{jk}(\overline{y_j}) \wedge C \wedge \bigcirc \Box \bigwedge_{i\in I} R_i \tag{14}$$

By renaming the variables in the tuples $\overline{y_j}$ to be unique across the k conjuncts, the existential quantification can be moved outwards over the conjunctions. Let $\overline{z_{jk}}$ denote the tuples of distinct variables. Furthermore, since C is independent of the variables in the tuples $\overline{z_{jk}}$, it can be moved into the quantifiers scope.

$$\mathcal{M}([]) \models (\exists\overline{z_{jk}}_{,j\in J,k\in K}.\ \bigwedge_{j\in J} \bigwedge_{k\in K} G_{jk}(\overline{z_{jk}}) \wedge C) \wedge \bigcirc \Box \bigwedge_{i\in I} R_i \tag{15}$$

Consider now the subformula $\bigwedge_{j\in J} \bigwedge_{k\in K} G_{jk}(\overline{z_{jk}}) \wedge C$. This is a future time formula and can be rewritten in a standard form which separates out present and strict future subformulae. For 15 to be satisfied, this form may be **true**, or a disjunction $\bigvee_{l\in L} f_l(\overline{u_l})$ where

$$\bigcup_{l\in L} \textbf{elems}\ \overline{u_l} = \bigcup_{j\in J} \bigcup_{k\in K} \textbf{elems}\ \overline{z_{jk}}$$

and each $f_l(\overline{u_l})$ is of either of the form

$\Lambda_l(\overline{u_l})$	a conjunction of literals
$\Lambda_l(\overline{v_l}) \wedge \bigcirc C_l(\overline{w_l})$	where **elems** $\overline{v_l} \cup$ **elems** $\overline{w_l}$ = **elems** $\overline{u_l}$
$\bigcirc C_l(\overline{u_l})$	

where $C_l(\ldots)$ is a future time formula. Assuming that the separated form is not **true**, the existential quantifiers $\exists\overline{z_{jk}}_{,j\in J,k\in K}$ can now be moved inwards over the disjunction, giving 16.

$$\mathcal{M}([]) \models (\bigvee_{l\in L} \exists\overline{u_l}\,.f_l(\overline{u_l})) \wedge \bigcirc \Box \bigwedge_{i\in I} R_i \tag{16}$$

For 16 to be satisfied, one of the disjuncts must be true. Without loss of generality, let this disjunct have index l. A binding α_l, for sake of argument, must also exist for the variables contained in the tuple $\overline{u_l}$. Thus 16 reduces to satisfying the following.

$$\mathcal{M}([]) \models \eta(\alpha_l, f_l(\overline{u_l})) \wedge \bigcirc \Box \bigwedge_{i\in I} R_i \tag{17}$$

By writing f_l in its general separated form, and letting Λ'_l denote $\eta(\alpha_l, \Lambda_l(\overline{v_l}))$ and C'_l denote $\eta(\alpha_l, C_l(\overline{w_l}))$, the following is obtained

$$\mathcal{M}([]) \models \Lambda'_l \wedge \bigcirc C'_l \wedge \bigcirc \Box \bigwedge_{i\in I} R_i \tag{18}$$

For this to be satisfied, the propositional part Λ'_l must be true in the current state (i.e. the first state of σ). Assume that this is the case. Now, by letting $\mathcal{M}([])'$ denote the model structure $(\sigma_h\hat{\ }[\textbf{hd}\ \sigma]\hat{\ }\textbf{tl}\ \sigma, [], |\sigma_h|+1)$, it is clear that the following must be satisfied.

$$\mathcal{M}([])' \models \Box \bigwedge_{i\in I} R_i \wedge C'_l \tag{19}$$

This is precisely the form with which we started. Hence we have derived a recursive mechanism which will construct models in a stepwise fashion.

Using this derivation, an execution mechanism for FML can be defined, see Figure 12 for an outline interpreter in Prolog-like notation. An explanation of the predicates used in the interpreter follows, emphasising the connection with the derivation given above.

- **check-past(rules, history, constraints, bindings)**
  As in PML, this predicate, when satisfied, has the effect of determining the activated rules from **rules**, i.e. those rules that have past-time antecedents satisfied on **history**, for particular bindings of variables. The variable **constraints** is then bound to a list of the future parts of the activated rules, and a list of bindings that apply to the universal variables in **constraints** is bound to **bindings**[9]. This

[9]Note that, as all universal variable bindings have been added to **bindings**, the only variables that appear in **constraints** and have no binding are existentially quantified.

```
execute([],[],history) :- true.

execute(rules,commitments, history) :-
        check-past(rules, history, constraints, bindings),
        rewrite(constraints, commitments, bindings, exec-form),
        choose(exec-form, rules, current-constraints, new-commitments),
        build-new-state(history, current-constraints, new-history),
        execute(rules, new-commitments, new-history).
```

Figure 12: A basic interpreter for First-Order METATEM.

process of generating the universal bindings corresponds to the transformation from formula 8 and 10 in the derivation.

- **rewrite(constraints, commitments, bindings, exec-form)**
  The **bindings** are used to expand the formulae in **constraints** (as in 11 to 16), which are then conjoined with those from **commitments** and rewritten into a formula in the standard form. This corresponds to the formula to be executed in the current state, given as the disjunction in 16, and is bound to **exec-form**.

- **choose(exec-form, rules, current-constraints, new-commitments)**
  The predicate **choose** has the effect of picking a disjunct to execute from **exec-form** and binding propositional constraints to **current-constraints** and next-state commitments to **new-commitments** (with the principal $\bigcirc$ stripped off). At this stage, not only are the bindings for unbound atoms chosen, but bindings are also chosen for any unbound existential variables. This procedure corresponds to the transformation from 16 to 18 in the derivation.

  Once a disjunct and binding has been chosen, the current state is fully constrained.

- **build-new-state(history, current-constraints, new-history)**
  This predicate has the same effect in the the case of FML as it did in PML: a new state is built in accordance with **current-constraints** and enviroment values are recorded. The formula represented by **current-constraints** is equivalent to Λ'_l in formula 18 of the derivation.

  Note that a loop-checking mechanism is again invoked at this point. If backtracking does not occur, the new state generated is appended to **history** to form **new-history**.

4.2.2 Example

As an example of the use of first-order METATEM, consider extending the resource controller given earlier to a greater number of processes. We could achieve this by having, as we do in the earlier rules, a different proposition for each request or allocation for each process, i.e.,

$$r_1,\ r_2,\ \ldots,\ r_n \qquad \text{and} \qquad a_1,\ a_2,\ \ldots,\ a_n$$

However, this is tedious, difficult to extend and clutters the program rules. We can represent the resource controller in first-order METATEM using the rules given in Figure 13.

In this example, x, x_1, and x_2 are universal variables, R is an environment predicate, A is a component predicate, and a, b, and c are ground terms. The set of atoms includes $R(a)$, $R(b)$, $R(c)$, $A(a)$, $A(b)$, and $A(c)$. A sample run of this resource controller is given in Figure 14.

The convention used in the example is that atoms that are not explicitly mentioned in the state are assumed to be false. Thus, $R(c)$ and $A(b)$ are false in state 0. Below, a more detailed description of the

$$\circledcirc R(x) \Rightarrow \Diamond A(x) \quad (1)$$

$$\neg R(x)\ \mathcal{Z}\ (A(x) \wedge \neg R(x)) \Rightarrow \neg A(x) \quad (2)$$

$$\circledcirc (A(x_1) \wedge A(x_2)) \Rightarrow (x_1 = x_2) \quad (3)$$

Figure 13: First-order METATEM rules for a Resource Manager

Environment	*Component*	*State*	*Commitments*
$R(a)$, $R(b)$		0	
$R(c)$	$A(a)$, $A(b)$	1	
In the process of building state 2, the execution mechanism finds inconsistencies such as $a = b$ and $b = a$. Consequently, the construction of state 2 is aborted and the execution backtracks to the choice point in state 1.			
$R(c)$	$A(a)$	1	$\Diamond A(b)$
	$A(b)$	2	$\Diamond A(c)$
	$A(c)$	3	

Figure 14: Sample Execution of Resource Manager

steps involved in the execution of the resource controller will be given. The states referred to are those outlined in the execution trace in Figure 14.

In the initial state (state 0), the environment sets $R(a)$ and $R(b)$ to be true and, by default, sets $R(t)$ to false for other terms t where $R(t) \in$ Atoms. As there is no previous state, the antecedent of rule 2 in the program rules is trivially satisfied for all x. Thus, $A(t)$ is set to false for other terms, t, where $A(t) \in$ Atoms. As the executable formula generated refers only to the current state, no commitments are carried forward.

In state 1, the environment sets $R(c)$ to be true, but sets $R(t)$ to be false for all other appropriate terms t. The antecedent of rule 3 is again unsatisfied as there was no binding of x for which $A(x)$ was true in the previous state. Also, the antecedent of rule 2 is unsatisfied for a and b (as $R(a)$ and $R(b)$ occurred in the last state), but is satisfied for all other bindings of x. This generates a commitment to make $A(t)$ false for all terms t, except a and b, where $A(t) \in$ Atoms.

During the construction of state 2 an inconsistency is found as the antecedent fo rule 3 is satisfied for $x_1 = a$ and $x_2 = b$ giving a commitment to make $a = b$ true (similarly with $x_1 = b$ and $x_2 = a$). As these is no way to make this true and as there are no choices that avoid trying to make it true, the execution mechanism backtracks to a choice point that occurred during the construction of state 1. The only choice made in state 1 was to execute the disjunct that set both $A(a)$ and $A(b)$ to be true. A different disjunct is then chosen that sets $A(a)$ to be true, $A(b)$ to be false and generates the commitment $\Diamond A(b)$. Thus $A(a)$ is set to true in state 1.

During the construction of state 2, no inconsistency is found as the antecedent of rule 3 is satisfied only for $x_1 = a$ and $x_2 = a$ and making $a = a$ true is not difficult. The environment sets $R(t)$ to be false for all terms t where $R(t) \in$ Atoms and the rules generate $\Diamond A(c)$, to execute along with the commitment $\Diamond A(b)$. At this stage, the disjunct that set both $A(b)$ and $A(c)$ to be true could have been chosen; if it had, backtracking would have occurred during the construction of state 3. We chose to show the execution of one of the satisfiable disjuncts and so $A(b)$ is set to true while $A(t)$ is set to false for all other appropriate terms t. This generates a commitment to satisfy $\Diamond A(c)$.

In state 4, the environment sets $R(t)$ to be false for all terms t where $R(t) \in$ Atoms and the rules generate the constraint that $A(t)$ must be false for all t except c. As there is commitment of $\Diamond A(c)$, carried over from the previous state, $A(c)$ is here set to true and no commitments are carried forward.

4.3 Meta-programming in METATEM

A further aspect of METATEM is its meta-programming capability. We will here only provide a flavour of this and, as a first example, will consider the outline of a simple meta-interpreter for METATEM given in Figure 15. This has been defined by three rules in the appropriate form. The first rule defines the initial

(1) $\bullet$**false** $\Rightarrow$ input(rules) $\wedge$
check-past(rules, satisfied-rules) $\wedge$
rewrite(satisfied-rules, [ ], exec-form) $\wedge$
choose(exec-form, current-constraints, new-commitments) $\wedge$
apply(current-constraints) $\wedge$
to-execute(rules, new-commitments)

(2) $\circledcirc$ to-execute(rules, commitments) $\Rightarrow$ check-past(rules, satisfied-rules) $\wedge$
rewrite(satisfied-rules, commitments, exec-form) $\wedge$
choose(exec-form, current-constraints, new-commitments) $\wedge$
apply(current-constraints) $\wedge$
to-execute(rules, new-commitments)

(3) $\bullet$**false** $\Rightarrow$ $\square$tick

Figure 15: A basic meta-interpreter for METATEM—M_1

state; the second rule defines an execution step of the interpreter; and finally, the third rule defines the execution to be infinite. The definition of the computation step (rule 2) is similar in form to the interpreter given in Section 2.4. However, the major difference is that the execution state, previously an argument of the execute predicate, is now implicit in the meta-interpreter execution.

One unusual aspect of this meta-interpreter is that the (conceptually) infinite tower of such interpreters are all running in step and in the same logical time. They are all running in step because the $\circledcirc$ operator in rule 2 forces each meta-interpreter to be in step with the interpreter on which it is being executed. They all run in the same logical time because, in the first execution state, rule 1 has the effect of reading in the new program rules (i.e. a new meta-interpreter) and executing them. Thus all the meta-interpreters begin execution from the same state!

Suppose now we wished to provide a METATEM interpreter that allows the dynamic modification of program rules. This would require producing a new meta-interpreter, say M_2, which has as its computation step the following rule.

$\circledcirc$ to-execute(rules, commitments) $\Rightarrow$ input-new-rules(rules, new-rules) $\wedge$
check-past(new-rules, satisfied-rules) $\wedge$
rewrite(satisfied-rules, commitments, exec-form) $\wedge$
choose(exec-form, current-constraints, new-commitments) $\wedge$
apply(current-constraints) $\wedge$
to-execute(new-rules, new-commitments)

The rules of M_2 are then used as the program rules for M_1, thus providing a new top level interpreter. Note that M_2 is in fact just another meta-interpreter, given that it too can run on top of an infinite tower of M_2's. This mechanism can also be used to change the future commitments between execution steps.

5 A General Framework

The METATEM approach can be considered as a special case of a more general computational problem involving quantifiers in Herbrand universes. To explain what we mean, consider the following example. We are given a database with the following items of data:

(1)	$Bad(O)$	"Oliver is bad"
(2)	$\forall x[Bad(x) \Rightarrow Bad(S(x))]$	"Any son of a bad person is also bad"
(3)	$\exists x Tall(x)$	"Someone is tall"
	our query is	
(4)	$\exists x[Tall(x) \wedge Bad(x)]$	"Is there someone both tall and bad?"

The answer to this query is negative. It does not follow from the database in first order classical logic. If we try to use a Horn Clause procedural computation we get:

Data

(1) $Bad(O)$

(2) $Bad(x) \Rightarrow Bad(S(x))$

(3) $Tall(c)$

Query

(4) $Tall(x) \wedge Bad(x)$

The c is a Skolem constant. The query will fail or loop.

Suppose we add the *extra assumption* 5 to the database, where:

(5) The existential quantifiers involved (i.e. the $\exists x$ in (3)) refer to the Herbrand universe generated by the function symbol $S(x)$.

This means that the constant c must be an element of the set $\{O, S(O), S^2(O), \ldots\}$. In this case, the query should succeed. There does exist an x which is both tall and bad. How can we logically put this extra input (5) into the database? Are we doomed to reasoning in such cases in the meta-language only?

The answer is that temporal logic does it for us. Let the "flow of time" be the set

$$\{O, S(O), S^2(O), \ldots\}$$

i.e. Oliver and his descendants are considered to be points of time. We let $S^m(O) < S^n(O)$ iff $m < n$.

We can read $Tall(x)$ (respectively $Bad(x)$) to mean $Tall$ (respectively Bad) is true at "time" x. The data and query now become

Data

(1*)	Bad	*Bad* is true at time O, which is "now"
(2*)	$\Box(Bad \Rightarrow \bigcirc Bad)$	Always if *Bad* is true then *Bad* is also true next
(3*)	$\Diamond Tall$	Sometime *Tall* is true

Query

(4*) $\Diamond(Tall \wedge Bad)$

We see that $\diamondsuit$ is read exactly as we want it and the simple database (1), (2), (3), (5) is properly logically represented by (1*), (2*), (3*).

We can now see what is the general framework involved. First we are dealing with first order logic with function symbols. Second we are dealing with special quantifiers, which we call *γ-quantifiers* (γ for "generated" universe quantifiers). These quantifiers have the form $\exists_{\{f_i\}}$, $\forall_{\{f_i\}}$, where $\exists_{\{f_i\}}A(x)$, is true iff there exists an element y generated in the Herbrand universe of the function symbols $\{f_i\}$, such that $A(y)$ is true. $\forall_{\{f_i\}}A(x)$ reads for all y in the Herbrand universe of the function symbols $\{f_i\}$, $A(y)$ is true.

The data and query can now be written in the new γ-quantifiers language as follows:

Data

(1**) $Bad(O)$

(2**) $\forall_{\{O,S\}}Bad(x)$

(3**) $\exists_{\{O,S\}}Tall(x)$

Query

(4**) $\exists_{\{O,S\}}(Tall(x) \wedge Bad(x))$

Our METATEM interpreter is nothing more than a model generating method for first order logic with γ-quantifiers for unary Skolem functions. We now give formal definitions.

Definition 5.1 *A γ-first order logic with unary functions consists of the following.*

1. *A first order language with variables $\{x_i\}$, predicates P_i, a stock of unary functions $\{f_i\}$, a stock of 0-functions (constants) $\{c_i\}$ and the usual classical connectives.*

2. *A stock of γ-quantifiers $\exists_\Gamma$ and $\forall_\Gamma$, indexed by finite sets $\Gamma = \{f_i, c_j\}$ of unary functions and constants. We assume that each such Γ contains at least one constant c_1.*

3. *Well-formed formulae are defined in the usual way*

Definition 5.2 *Given a set Γ as in definition 5.1, the Herbrand universe generated by Γ, denoted by H_Γ, is the smallest set containing the constants of Γ which is closed under the application of the functions of Γ, i.e. satisfying*

$$\text{if } c_i \in \Gamma \text{ then } c_i \in H_\Gamma \text{ and,}$$
$$\text{if } x \in H_\Gamma \text{ and } f \in \Gamma \text{ then } f(x) \in H_\Gamma.$$

Definition 5.3 *For any Γ, the binary relation N_Γ on the Herbrand universe, H_Γ, is defined by*

$$t_1 N_\Gamma t_2 \;\text{ iff }\; t_2 = f(t_1)$$

where $f \in \Gamma$ is a unary function symbol. Define $<_\Gamma$ as the transitive closure of N_Γ.

Notice that if $\Gamma \subseteq \Gamma'$ then $<_\Gamma \subseteq <_{\Gamma'}$. For example, $x < f(x) < ff(x)$.

Definition 5.4 *A γ-structure for a γ-first order logic has the form (D, h, R), where D is the domain ($D \neq \emptyset$), h is an assignment and R is a transitive relation on D, satisfying the following conditions:*

1. *if x is a constant $c \in \Gamma$ then $h(c) \in D$,*

2. *if f is a unary function symbol then $h(f)$ is a unary function on D such that for all $d \in D$, $d\,R\,h(f)(d)$ holds,*

3. *h is a Herbrand assignment, i.e. the following hold for any $c, d, f, g \in \Gamma$:*

   (a) *if $c \neq d$ then $h(c) \neq h(d)$*

(b) *if* $f \neq g$ *then* $\forall x \in D(h(f)(x) \neq h(g)(x))$

(c) *if* $x \neq y$ *then* $h(f)(x) \neq h(f)(y)$

The definition of truth in such a model is the same as in the classical case, with the following clauses added for the γ-quantifiers.

$$\begin{array}{lll} h \models \forall_\Gamma A(x) & \text{iff} & \text{for all } x \in H_\Gamma, h \models A(x) \\ h \models \exists_\Gamma A(x) & \text{iff} & \text{for some } x \in H_\Gamma, h \models A(x) \end{array}$$

For example, consider a language with one constant O and two function symbols f_1 and f_2. Thus $\Gamma = \{O, f_1, f_2\}$. The Herbrand universe H_Γ is $\{O, f_1(O), f_2(O), f_1 f_2(O), f_2 f_1(O), \ldots\}$. The natural ordering on the Herbrand universe is $<$ ($<_{\{O,f_1,f_2\}}$). (H_Γ,$<_\Gamma$) can be viewed as a binary temporal tree. The quantifier $\exists_\Gamma$ corresponds to $\Diamond$.

Definition 5.5 *(Γ-specification)*
Let there be given a first order γ-quantifier language. A database is said to be a Γ-specification database if it contains clauses of the form $\forall_\Gamma x_i[A(x_i) \rightarrow B(x_i)]$ where A is a formula with free variables x_i. The quantifiers in A all have the form γ-quantifiers restricted to be $< x_i$. The quantifiers of B are of the form

$$Q_\Gamma(x, y) = \exists_\Gamma y > x_i \forall_\Gamma u(x_i < u < y)$$

$A(x_i)$ corresponds to querying the past and $B(x_i)$ corresponds to specifying the future.

We are now ready to explain the general framework in which the METATEM interpreter resides. METATEM is a method for generating Herbrand γ-models for γ-specification first order databases.

Example — Linear Time Temporal Logic Thus, the PML interpreter described above is an interpreter for a language $\Gamma = \{O, S\}$, with one constant and one unary function. It utilises specific properties of this Γ (flow of time).

Example — Branching Time Temporal Logic There exist temporal logics such as CTL [CES86] that contain connectives of the form $\overline{\Diamond}$, which operate on the Herbrand universe of a set Γ of functions and have semantics:

$$\begin{array}{lll} \overline{\Diamond} A \text{ is true at } t & \text{iff} & \text{for all paths } \pi \text{ through } t, \text{ there exists an } x \text{ such that } x > t \text{ and } x \in \pi \\ & & \text{and } A \text{ is true at } x. \end{array}$$

Thus for example, if $\Gamma = \{f_1, \ldots, f_k\}$, $\overline{\Diamond} A$ is true at t iff for every path π with $t \in \pi$, there exists an $x \in H_\Gamma \cap \pi$, $t <_\Gamma x$, in which A is true. In CTL, this operator is represented by the pair AF.

Since H_Γ is a Herbrand universe, there is a finite set, S_A, of points in which A is true such that

$$\forall x \exists y.\ x \leq y \wedge y \in H_A.$$

Let $\mathbf{Fin}_A$ be the connective saying that A is true at a finite number of points of the Herbrand universe. Then $\overline{\Diamond}$ is essentially inter-definable using **Fin** as follows:

Definition 5.6 *Let Γ be a set of unary functions and c a constant. A path π in H_Γ is a sequence (t_i) such that $t_i N_\Gamma t_{i+1}$*[10]*. Consider the following connectives ($[A]_t$ reads A is true at t):*

$[\Diamond_\Gamma A]_t = 1$ *iff* $\exists x \in H_\Gamma.\ t <_\Gamma x \wedge [A]_x = 1$

$[\blacklozenge_\Gamma A]_t = 1$ *iff* $\exists x \in H_\Gamma.\ x <_\Gamma t \wedge [A]_x = 1$

[10]Note that set membership, $\in$, is extended in the obvious way to sequence membership.

$[\overline{\Diamond}_\Gamma A]_t = 1$ *iff* $\forall \pi. \, (\pi$ *is a path in* $H_\Gamma \wedge t \in \pi) \Rightarrow (\exists x. \, x \in \pi \wedge t <_\Gamma x \wedge [A]_x = 1)$

$[\mathbf{Fin}_\Gamma A]_t = 1$ *iff the set of* x *such that* $t <_\Gamma x$ *and* $[A]_x = 1$ *is finite*

Let $\Box = \neg\Diamond\neg$, $\blacksquare = \neg\blacklozenge\neg$ and $\overline{\Box} = \neg\overline{\Diamond}\neg$.

Proposition 5.1 *For* Γ *finite, we have:*

$$(k \wedge \blacksquare_\Gamma \neg k \wedge \Box_\Gamma \neg k) \;\Rightarrow\; (\overline{\Diamond}_\Gamma A \Leftrightarrow \Box_\Gamma \Diamond_\Gamma \blacklozenge_\Gamma (A \wedge \blacklozenge_\Gamma k) \wedge \mathbf{Fin}\Diamond_\Gamma (A \wedge \blacksquare_\Gamma (\blacklozenge_\Gamma k \Rightarrow \neg A)))$$

Proof: The above says that if k is true exactly at the point t that $\overline{\Diamond}_\Gamma A$ is equivalent to saying that

1. every point has a higher point somewhere in whose past, after the point at which k is true, A is true and

2. the set of points above t in which A first becomes true is finite.

Clearly there is equivalence over the Herbrand universe H_Γ.

Thus the general framework for our temporal language is model building for the quantifiers $\forall_\Gamma$, $\exists_\Gamma$, and **Fin**.

Definition 5.7 *The execution predicate for* $\overline{\Diamond}$ *and* $\Diamond$ *are as follows:*

1.

$$\mathbf{exec}(\overline{\Diamond}_\Gamma A) = \bigwedge_{i=1}^{m} \mathbf{exec}(\bigcirc_i A \vee \bigcirc_i \overline{\Diamond}_\Gamma A)$$

2.

$$\mathbf{exec}(\Diamond_\Gamma A) = \bigvee_{i=1}^{m} \mathbf{exec}(\bigcirc_i A \vee \bigcirc_i \Diamond_\Gamma A)$$

where $\Gamma = \{c, f_1, \ldots, f_m\}$ *and* $\bigcirc_i A$ *is true at* t *iff* A *true at* $f_i(t)$*.*

Note that proposition 5.1 depends on the fact that Γ is finite. If we have an infinite number of successor functions, the connective $\overline{\Diamond}$ is independent. We can regard $\overline{\Diamond}$ as a special second order quantifier on the Herbrand universe.

5.1 γ two sorted predicate logic

We are now ready to extend the γ-quantifier approach to predicate logic. The extension is simple. We imagine many sorted predicates $P(t, x_i, \ldots, x_n)$ defined over two domains. t ranges over a Herbrand universe H_Γ and $x_i, \ldots, x_n$ range over an ordinary set D. The language contains the usual connectives and the two sets of quantifiers — $\forall x$ and $\exists x$ for the $\{x\}$ sort and $\forall_\Gamma t$ and $\exists_\Gamma t$ for the t sort. This language defined the γ-predicate logic. It corresponds to ordinary temporal predicate logic.

Definition 5.8 *A* γ *two sorted predicate logic consists of the following:*

1. *A two sorted language with variables* $\{t_i\}$ *for the first sort and* $\{x_i\}$ *for the second sort, predicates of the form* $P(t, x_i, \ldots, x_n)$ *where* $n \geq 0$ *and* t *is the one sort of variable and the* x_i *are the other sort, a stock of unary functions and a stock of constants* c_i *for generating the terms of the first sort.*

2. *A stock of* Γ *quantifiers for the first sort (as in definition 5.1) and the usual quantifiers* $\forall$ *and* $\exists$ *for the second sort.*

3. *well-formed formulas are defined as usual for the two-sorted predicate logic.*

4. *A* γ *two sorted structure has the form* (D, h, R, g, U)*, where* U *is the universe of the* $\{x_i\}$ *sort,* (D, h, R) *is as in definition 5.4 and* g *is an assignment function, assigning for each* $n+1$ *predicate symbol* P *and each* $t \in D$ *a subset* $g(t, P) \subseteq D^n$*.*

The definition of truth in such a model is the same as the classical one on $\forall x$, $\exists x$ and the γ-definition and conditions (of 5.6) on the $\forall_\Gamma t$, $\exists_\Gamma t$.

A Proofs of Theorems

Here we present outline proofs for Theorems 3.1 and 3.2. First we establish the following Lemmas.

Lemma A.1 *Given a set of rules $\{R_i \mid i \in I\}$, each of the form $\xi_i \Rightarrow \psi_i$ where $\xi_i \in Wff_<$ and $\psi_i \in Wff_\geq$, together with a commitment C ($\in Wff_\geq$), and a finite history state sequence σ_h,*

$$\textbf{if } \textit{execute}(\bigwedge_{i\in I} R_i, C, \sigma_h) \textit{ fails } \textbf{then } \textit{there is no extension } \sigma \textit{ such that } \sigma_h\hat{}\sigma, |\sigma_h| \models (\Box \bigwedge_{i\in I} R_i) \wedge C$$

Outline Proof We use computation induction.
First note that *execute* calls *check-past* to reduce the formula to be executed, namely

$$\Pi : \qquad (\Box \bigwedge_{i\in I} R_i) \wedge C$$

to the formula

$$\Pi' : \qquad (\bigwedge_{j\in J} \psi_j) \wedge C \wedge \bigcirc(\Box \bigwedge_{i\in I} R_i)$$

where

$$R_i \;=\; \xi_i \Rightarrow \psi_i \quad \text{and} \quad J \;=\; \{i \mid \exists\sigma.\sigma_h\hat{}\sigma, |\sigma_h| \models \xi_i{}^{11}\;,\; i \in I\}$$

such that

$$\exists\sigma.\sigma_h\hat{}\sigma, |\sigma_h| \models \Pi \quad \text{iff} \quad \exists\sigma.\sigma_h\hat{}\sigma, |\sigma_h| \models \Pi'$$

The variable *satisfied-rules* of *check-past*, in fact returns the formula $\bigwedge_{j\in J} \psi_j$.
Next *execute* uses *rewrite* to reduce Π' to a logically equivalent form, separating out the present,

$$\Pi'' : \qquad (\bigvee_{k\in K} G_k) \wedge \bigcirc \Box(\bigwedge_{i\in I} R_i)$$

where

$K \neq \emptyset$: each G_k is either of the form $\varphi_k \wedge \bigcirc\psi_k$ or of the form φ_k, where $\varphi_k \in Wff_=$ is propositionally consistent and $\psi_k \in Wff_\geq$,

$K = \emptyset$: Π'' is **false**.

In the case of K being empty, *rewrite*$(\ldots)$ fails, causing failure of *execute*$(\bigwedge_{i\in I} R_i, C, \sigma_h)$. Clearly there is no extension to σ_h that will be a model for **false**.
In the case of K being non-empty, *choose*$(\ldots)$ will fail, causing subsequent failure of

$$\textit{execute}(\bigwedge_{i\in I} R_i, C, \sigma_h)$$

if for every $k \in K$ either

1. for every state s, total wrt the proposition alphabet, satisfying φ_k, the recursive call of *execute* on the history extended by state s fails, i.e. *execute*$(\bigwedge_{i\in I} R_i, \psi_k, \sigma_h\hat{}[s])$ fails, or

2. ψ_k contains an unsatisfiable eventuality with respect to the history σ_h and rules R_i.

   As mentioned previously this critical condition can be detected in a number of ways. The prototype [Fis88] uses a loop check mechanism described in Section 3.3.

Again, these are precisely the cases when there is no extension to σ_h that will yield a model for Π. Note that 1 above is the computation induction step.

Lemma A.2 *Given a set of rules $\{R_i \mid i \in I\}$, each of the form $\xi_i \Rightarrow \psi_i$ where $\xi_i \in Wff_<$ and $\psi_i \in Wff_\geq$, together with a commitment C ($\in Wff_\geq$), and a finite history state sequence σ_h,*

if *execute*$(\bigwedge_{i\in I} R_i, C, \sigma_h)$ *does not fail*

then *its infinite* (ω) *execution will generate an infinite* (ω) *extension to* σ_h.

[11] Remember that ξ_i is a pure past time formula and only depends on the real history, σ_h

Outline Proof Consider an infinite, non-failing, run of *execute*($\bigwedge_{i\in I} R_i$, C_0, σ_h). The run can viewed as an exploration of the following computation tree.

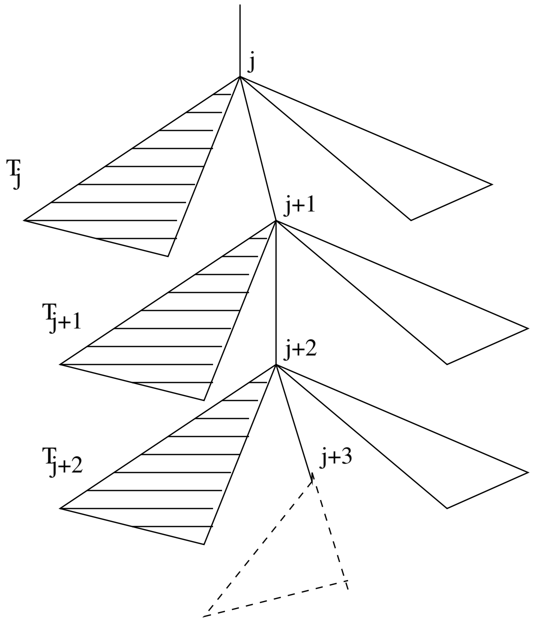

Figure 16: Partial Computation Tree

The computation starts at node 0 with a call of *execute*($\bigwedge_{i\in I} R_i$, C_0, σ_h). Each, actual, node in the computation tree is generated by some recursive call of *execute*. At each node, *execute* has a finite number of potential ways to recurse, each way corresponding to execution of one of the disjuncts from the formula to be executed at that point. Some recursive calls of *execute* may fail and backtracking to an earlier choice point, i.e. node, occurs. Consider node j. The shaded, finite, tree branching from the left of node j corresponds to n_j first level recursive calls of *execute* , each of which failed and caused a backtrack to node j. The unshaded tree branching to the right of node j represents potential unexplored computation tree. The branching at any node j in this tree is finite and matches with the number of choices that exists from the expansion of the formula to be executed at that node into the disjunctive form $\bigvee_i^{n_j} f_{j_i}$ (cf. Section 2.4). By ensuring that *execute* never explores any branch from a node more than once, an infinite (ω), backtracking, run of *execute* explores the tree to an infinite (ω) depth. Consider an ω-sequence of nodes "visited" by *execute* during its backtracking, but infinite, run. For example,

$$\ldots, j, \underbrace{\ldots\ \ldots,}_{T_j} j+1, \underbrace{\ldots\ \ldots,}_{T_{j+1}} j+2, \underbrace{\ldots\ \ldots,}_{T_{j+2}} j+3, \underbrace{\ldots\ \ldots,}_{T_{j+3}} \ldots$$

The subsequences denoted by T_j, etc, correspond to nodes constructed during failing calls of *execute* from node j, i.e. the shaded trees, T_j in Figure 16. Of interest is the projection of nodes $j, j+1, \ldots$. Each recursive call of *execute*, say to node $j+1$ from node j, extends the previous history argument σ_j by one state s_j. The ω-extension implicitly generated by *execute* is defined as the sequence of states $s_0, \ldots, s_j, s_{j+1}, \ldots$.

Lemma A.3 *Given a set of rules* $\{R_i \mid i \in I\}$, *each of the form* $\xi_i \Rightarrow \psi_i$ *where* $\xi_i \in Wff_<$ *and* $\psi_i \in Wff_\geq$, *together with a commitment* C ($\in Wff_\geq$), *and a finite history state sequence* σ_h,

> **if** *execute*($\bigwedge_{i\in I} R_i$, C, σ_h) *does not fail*
>
> **then** *there is an extension* σ *generated by execute such that* $\sigma_h\hat{}\sigma, |\sigma_h| \models \Box(\bigwedge_{i\in I}) \wedge C$

Outline Proof If the computation of *execute* does not fail then, by Lemma A.2 it will recurse infinitely generating an infinite extension to σ_h, namely the sequence $\sigma = s_0, s_1, s_2, \ldots$, where the states s_i, etc, are, in fact, total with respect to the proposition alphabet. It is easy to see that, by the construction process applied by *execute* (*build-new-state*), the extension σ will yield a model in the cases that the commitment argument of *execute* never contains a $\Diamond$-formula. We argue that a model is also obtained in the case that the commitment contains a $\Diamond$-formula. If such $\Diamond$-formula is never satisfied, then by definition it will be continually postponed for satisfaction in some future moment, i.e. will always be carried forward in the commitment argument. Since the number of possible distinct states is finite, repetition must occur and hence a loop carrying an unsatisfied $\Diamond$-formula will be detected and failure of *execute* will occur. But *execute* does not fail, therefore, the extension it produces must also satisfy all $\Diamond$-formulae that may occur.

The proof of Theorem 3.1 and 3.2 is now trivial.

Theorem 3.1 Let P be a program consisting of a set of program rules $\{R_i \mid i \in I\}$, each of the form $\xi_i \Rightarrow \psi_i$ where $\xi_i \in \mathit{Wff}_<$ and $\psi_i \in \mathit{Wff}_\geq$. If the program formula, f, defined as $\Box \bigwedge_{i\in I} R_i$ is satisfiable then *execute*($\bigwedge_{i\in I} R_i$, **true**, []) will generate a model for f.

Proof First note that if $\Box \bigwedge_{i\in I} R_i$ is satisfiable, then is some extension σ to the empty sequence, [], such that $[]\hat{}\sigma, 0 \models (\Box \bigwedge_{i\in I} R_i) \wedge \mathbf{true}$. Therefore by Lemma A.1, *execute*($\bigwedge_{i\in I} R_i$, **true**, []) does not fail. By Lemma A.3, as *execute* does not fail, it will generate an infinite extension σ such that $[]\hat{}\sigma, 0 \models (\Box \bigwedge_{i\in I} R_i) \wedge$ **true**, i.e. $\sigma, 0 \models (\Box \bigwedge_{i\in I} R_i)$

Theorem 3.2 Let P be a program consisting of a set of program rules $\{R_i \mid i \in I\}$, each of the form $\xi_i \Rightarrow \psi_i$ where $\xi_i \in \mathit{Wff}_<$ and $\psi_i \in \mathit{Wff}_\geq$. If *execute*($\bigwedge_{i\in I} R_i$, **true**, []) does not fail, then the program formula f, defined as $\Box \bigwedge_{i\in I} R_i$ is satisfiable and *execute* generates a model for f.

Proof Follows directly from Lemma A.3.

B Automata-Theoretic aspects of METATEM execution

If METATEM programs contain no references to environment variables then we can view the execution mechanism as tracing out a subtree of a semantic tableau associated with the program rules. This special construction is driven by the future formulae of the rules, the past formulae only being evaluated at existing nodes, with further nodes being introduced without gaps.

Thus, if the program consists of a set of METATEM rules $R_1, \ldots, R_n$ then the associated tableau is that for the program formula

$$\Box \bigwedge_{i=1}^{n} R_i$$

If we construct a Hintikka Graph[12] for such a formula (the graph will be finite due to the finite model property exhibited by this PML) we can then describe an infinite tree representing all the paths through this graph. Even if we assume that the graph construction filters out all propositionally inconsistent and next-time inconsistent sequences, some of the paths through the infinite tree may still be inconsistent. This is because eventualities (i.e., commitments generated by $\Diamond$ and $\mathcal{U}$ formulae) may be generated but never satisfied on certain branches. In the tableau method for temporal logic, the Hintikka Graph is usually 'marked' in such a way that no such sequences can be generated as models[13]. We can see this as pruning of the infinite tree, removing all paths that contain continually unsatisfied eventualities. This tree can be seen as the 'satisfiable subtree' of the original tree in which no checks for inconsistencies have been carried out.

[12]For an overview of the tableau method in propositional temporal logics, see Gough [Gou84] and Wolper [Wol85] and for an automata-theoretic view of this process, see [VW86].

[13]A Hintikka graph marked in this way is equivalent to a Büchi automaton.

The execution mechanism proceeds by exploring a portion of the tree that can be generated from a Hintikka Graph in an attempt to follow an infinite branch of the satisfied subtree described above. Though unsatisfiable branches may be followed for a finite number of states, the invocation of the loop-checking mechanism will force the execution mechanism to backtrack down such branches and explore different branches. Eventually, the execution mechanism will follow an infinite branch of the satisfied subtree. Once it has chosen such a path it will never backtrack past the choice point again. Though a branch that goes from the satisfied subtree back out into the unsatisfiable part of the original tree can be followed for a finite number of states, backtracking will eventually occur and exploration of the satisfied subtree will continue.

The soundness of the execution mechanism is given in such a way in Theorem 3.1.

Thus, METATEM programs can be 'compiled' into automata, i.e, those automata generated from Hintikka Graphs. A similar approach to the 'compilation' of METATEM program rules into automata is that of the synthesis of programs from reactive system specifications described by Pnueli and Rosner [PR89b]. As part of their work on the synthesis of reactive modules they synthesize, for finite systems with no environment interaction, Streett tree automata that are similar to the automata described above.

References

[Aba87a] M. Abadi.
Temporal-Logic Theorem Proving.
PhD thesis, Department of Computer Science, Stanford University, March 1987.

[Aba87b] M. Abadi.
The Power of Temporal Proofs.
In *Proc. Symp. on Logic in Computer Science*, pages 123–130, Ithaca, June 1987.

[AM85] M. Abadi and Z. Manna.
Non-clausal Temporal Deduction.
Lecture Notes in Computer Science, 193:1–15, June 1985.

[AM86] M. Abadi and Z. Manna.
A Timely Resolution.
In *Proc. Symp. on Logic in Computer Science*, pages 176–186, Boston, June 1986.

[Bau89] M. Baudinet.
Temporal Logic Programming is Complete and Expressive.
In *Proceedings of ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.

[Bel77] N. D. Belnap.
A Useful Four-Valued Logic.
In J. M. Dunn and G. Epstein, editors, *Modern Uses of Multiple-Values Logic*, pages 5–37. D. Reidel Publishing Company, 1977.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla.
Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic specifications.
ACM Transactions on Programming Languages and Systems, 8(2):244–263, January 1986.

[Fis88] M. D. Fisher.
Implementing a Prototype METATEM Interpreter.
Temple Group Report, Department of Computer Science, University of Manchester, November 1988.

[Gab87] D. Gabbay.
Modal and Temporal Logic Programming.

In A. Galton, editor, *Temporal Logics and their Applications*. Academic Press, London, December 1987.

[Gab89] D. Gabbay.
Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems.
In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of Colloquium on Temporal Logic in Specification, Altrincham, 1987*, pages 402–450. Springer-Verlag, 1989.
LNCS Volume 398.

[Gou84] G. D. Gough.
Decision procedures for Temporal Logic.
Master's thesis, Department of Computer Science, University of Manchester, October 1984.

[Hal87] R. Hale.
Temporal Logic Programming.
In A. Galton, editor, *Temporal Logics and their Applications*, chapter 3, pages 91–119. Academic Press, London, December 1987.

[HNNT86] T. Hattori, R. Nakajima, N. Niide, and K. Takenaka.
RACCO: A Modal-Logic Programming Language for Writing Models of Real-time Process-Control Systems.
Technical report, Research Institute for Mathematical Sciences, Kyoto University, November 1986.

[LP85] O. Lichtenstein and A. Pnueli.
Checking that Finite State Concurrent Programs Satisfy their Linear Specification.
In *Proceedings of the Twelfth ACM Symposium on the Principles of Programming Languages*, pages 97–107, New Orleans, Louisiana, January 1985.

[MM84] B. Moszkowski and Z. Manna.
Reasoning in Interval Temporal Logic.
In *AMC/NSF/ONR Workshop on Logics of Programs*, pages 371–383, Berlin, 1984. Volume 164, LNCS, Springer-Verlag.

[Mos83] B. Moszkowski.
Resoning about Digital Circuits.
PhD thesis, Department of Computer Science, Stanford University, July 1983.

[Mos86] B. Moszkowski.
Executing Temporal Logic Programs.
Cambridge University Press, Cambridge, England, 1986.

[MW84] Z. Manna and P. Wolper.
Synthesis of Communicating Processes from Temporal Logic Specifications.
ACM Transactions on Programming Languages and Systems, 6(1):68–93, January 1984.

[PR89a] A. Pnueli and R. Rosner.
On the Synthesis of a Reactive Module.
In *Proceedings of the 16th ACM Symposium on the Principles of Programming Languages*, pages 179–190, 1989.

[PR89b] A. Pnueli and R. Rosner.
On the Synthesis of an Asynchronous Reactive Module.
In *Proceedings of the 16th International Colloquium on Automata, Languages and Programs*, 1989.

[Sak86] T. Sakuragawa.
Temporal Prolog.
Technical report, Research Institute for Mathematical Sciences, Kyoto University, 1986.

to appear in *Computer Software*.

[VW86] M.Y. Vardi and P. Wolper.
Automata-Theoretic Techniques for Modal Logics of Programs.
Journal of Computer and System Science, 32(2):182–21, April 1986.

[Wol85] Pierre Wolper.
The Tableau Method for Temporal Logic: An overview.
Logique et Analyse, 110–111:119–136, June-Sept 1985.

[Zhi83] Tang Zhisong.
Toward a Unified Logic Basis for Programming Languages.
In R. E. A. Mason, editor, *Information Processing 83*, pages 425–429, Amsterdam, 1983. IFIP, Elsevier Science Publishers B.V. (North Holland).