Meta-Reasoning in Executable Temporal Logic

Howard Barringer **Michael Fisher**
Department of Computer Science,
University of Manchester,
Oxford Road,
Manchester M13 9PL, UK.
{howard,michael}@cs.man.ac.uk

Dov Gabbay **Anthony Hunter**
Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate,
London SW7 2BZ, UK.
{dmg,abh}@doc.ic.ac.uk

Abstract

Temporal logic can be used as a programming language. If temporal formulae are represented in the form of an implication where the antecedent refers to the past, and the consequent refers to the present and the future, then formulae that have their antecedent satisfied can be satisfied by considering the consequent as an imperative to be obeyed. Such a language becomes a natural alternative to programming languages, such as Prolog, for temporal reasoning. Here, the approach is extended to include an executable meta-language. This is of importance in developing interpretation, debugging, loop-checking, and simulation tools, and provides a representation for providing control knowledge in the execution of planning and scheduling programs.

1 Executable Temporal Logic

The traditional view of temporal logic is that it represents declarative statements about the world, or about possible worlds over time. Such statements relate the truth of propositions in the past, present and future, and so have been found useful in the representation of time-dependent systems (Pnueli, 1977; Allen and Hayes, 1985). Using this declarative approach, time is viewed from the *outside* — it is seen in its entirety. An alternative view of temporal logic is to consider it in terms of a declarative past and an *imperative* present and future, based on the intuition that a statement about the future can be imperative, initiating steps of action to ensure it becomes true. Using this view, time is seen from the *inside* — we are at a moment in time with the past behind us and the future yet to happen, but with 'instructions' about what to do in the future. In such an approach, the future must be actually made to happen, rather than just stating its properties. It is this second approach that we follow.

Temporal specifications can be given, naturally, as a collection of rules of the form

ANTECEDENT ABOUT THE PAST
$\Rightarrow$
CONSEQUENT ABOUT THE PRESENT AND FUTURE

For example, if the flow of time is represented as discrete linear order, e.g. the natural numbers, then atomic propositions are true or false at points on that line. Executable temporal logic statements can then be executed at a point X on that line by reviewing which atomic propositions were true in the past, i.e. at points less than X, and then taking steps to ensure the satisfication of the specification. If any of the necessary steps include an action in the present then, to ensure satisfaction of the specification, that action must be undertaken now. If any of the necessary steps include an action in the future then this instruction must passed on to the future.

Executable temporal logics of this form that have been developed include USF (Gabbay, 1989) and METATEM (Barringer et al., 1989; Fisher, 1989). Such logics are based on *US logic*[1] and incorporate a notion of interaction with the environment of the program. Predicates and propositions are partitioned into those controlled by the *component*, i.e. the program, and those controlled by the *environment*, i.e. the context of the program. Thus, at certain times in the execution of such a temporal formula, the environment may need to be consulted in order to provide the values of those atoms controlled by the environment.

If specifications are given in the above rule-based form, then, at any moment in time, the past can be checked and the future-time consequents of all the rules whose antecedents are satisfied can be collected together. These constraints on current and future behaviour can be rewritten as a formula of the form

$$\bigvee_i (\bigwedge_{j_i} l_{j_i} \wedge \bigcirc \varphi_i)$$

where each l_{j_i} is a literal and each φ_i is a present and future-time temporal formula. Thus, at every moment in time, the

[1] US logic is a temporal logic with until ('$\mathcal{U}$') and since ('$\mathcal{S}$') as its only temporal operators.

executing agent must endeavour to satisfy the above formula. It can do this by making any one of the disjuncts true. Choosing which disjunct to make true is a subtle problem, and this choice will depend on several factors, such as any commitments to satisfy other clauses, possible future deadlocks, and the environment at that point in time. There may be more than one valid choice at each time, and the actual choice may vary over time.

All formulae of propositional and first-order US-logic can be rewritten into the above executable form (Gabbay, 1989), although the executable form is not, in general, unique, i.e. a specification can be written into more than one equivalent form. For the propositional case, the execution is decidable, i.e. if a US-logic program is satisfiable, then a model for the program can be found through execution. First-order US-logic is, in general, not decidable, hence for applications where it is crucial to find a model if the program is satisfiable, we can either adopt a decidable subset of first-order US-logic, or harness heuristics that ameliorate the problem.

A METATEM execution mechanism has been defined for temporal logic programs that returns a model of the program if the program is satisfiable (Barringer et al., 1989). This result is conditional on the program comprising only component propositions. If environment atoms are introduced into a program, then branching time logic may be required in order to represent all possible environments (Clarke and Emerson, 1982; Pnueli and Rosner, 1989).

Executable temporal logic provides a natural alternative to programming languages, such as Prolog, for temporal reasoning. A temporal logic program given in our executable form provides a clear link between a declarative past and an imperative present and future, and is being developed for applications in planning, scheduling and process control. Though an executable temporal logic that incorporates the full generality of temporal logics can be inefficient, restricted temporal languages have been shown to be useful in many areas (Moszkowski, 1986; Fujita et al., 1986; Hattori et al., 1986; Caspi et al., 1987; Abadi, 1987).

The executable temporal logic considered in this report is one of the family of METATEM languages (Barringer et al., 1989). Other members of this family have been implemented (Fisher, 1989; Owens, 1990) and such languages have been used successfully in large case studies, for example the development of a PAYE ('Pay As You Earn') tax system (Torsun and Manning, 1990).

In this paper, we focus on an executable meta-language. We indicate why a meta-language is of interest, outline the logical basis of a meta-language for executable temporal logic, and consider applications in knowledge representation and reasoning.

2 Why Use A Meta-Language?

In (Barringer et al., 1989), we introduced propositional and first-order temporal logics (PML and FML) as instances of METATEM. We suggested that a rather different logic is required to carry out meta-level reasoning within METATEM. By meta-level reasoning, we mean the ability, in one language (the *meta-language*), to name and reason about statements from another language (the *object-language*). In the case of METATEM we require one (logical) language to be both the meta-language and the object-language, i.e., a language that can act as its own meta-language. Of course, this is not new in its own right, for example in Prolog meta-level reasoning about clauses can be carried out within the same language (Hill and Lloyd, 1988; van Harmelen, 1988; Nilsson and Małuszyński, 1990).

Using a first-order temporal logic, such as FML (Barringer et al., 1989), a wide variety of temporal specifications can be constructed and represented as METATEM programs. However, from a computational point of view greater expressive power is needed, as there is a requirement to be able to interact with, and to influence, a running METATEM process through meta-level manipulation.

Many useful programming tools are meta-level tools, for example, compilers, debuggers, partial evaluators, etc. Also, in declarative languages, many techniques for improving efficiency can be implemented by meta-level manipulation, for example program transformation techniques such as folding and unfolding. From a logical point of view, we can also see the benefit of meta-level statements as they give us the ability, for example, to change the logical basis of the interpreter. A particularly useful example of this is the ability to represent control knowledge to restrict or prioritize the possible execution paths (as for example in simulation or scheduling tasks). Thus, we require a logic that has some way of representing its own formulae in its domain of discourse, in particular, a logic for METATEM in which METATEM program rules can be represented.

There are several standard methods for developing meta-languages. One of the most widespread is the use of *quoting*. This technique, proposed by Frege and further developed by many others (Perlis, 1985), gives a way of producing a term in the logic that represents or 'names' a formula in the logic. Thus, if we have the formula $p(x) \wedge q(x)$, a new term representing the formula is given by "$p(x) \wedge q(x)$". As predicates are applied to terms of the language, they can obviously be applied to these 'naming terms', for example $r(\text{“}p(x) \wedge q(x)\text{”})$.

Thus, we can write the following type of formula in a temporal logic incorporating such a quoting mechanism[2],

$$\bullet p(\text{“}x \wedge y\text{”}) \;\Rightarrow\; q(\text{“}x\text{”}) \wedge r(\text{“}y\text{”}).$$

In such a way a logic capable of meta-level reasoning can be developed. In the case of METATEM, such a logic,

[2]Here, we use a discrete model of time and use '$\bullet$' as the operator representing 'at the last moment in time'.

called FML*, based on quoting has been developed (Fisher, 1990). The questions of the reduction of 'naming' terms to their constituent parts, the quoting of variables, and the scope of quantification are discussed elsewhere (Perlis, 1985; Fisher, 1990). Furthermore, the FML* approach corresponds to the *ground representation* described in (Hill and Lloyd, 1988).

In this report we will present an alternative approach which unifies object-level and meta-level into a single language without using explicit quoting. In our logic, called Meta-METATEM Logic, the set of terms is partitioned into object-level terms and meta-level terms. Object-level terms are the basic terms of the language; meta-level terms are terms that represent (or name) formulae in the language.

Though a technical definition of Meta-METATEM Logic (MML) is given in appendix A, a brief outline of the principles underlying MML will be given in the next section.

3 Meta-METATEM Logic (MML)

MML is a first-order temporal logic, based on a linear, discrete model of time. MML contains the standard temporal operators, such as 'sometime in the future' ('$\Diamond$'), 'always in the future' ('$\Box$') and 'at the next moment in time' ('$\bigcirc$'), along with their past-time counterparts, and uses a non-rigid form of quantification. A semantics for MML based on possible-worlds is given in appendix A and is outlined below.

In MML the domain over which terms range has been extended to incorporate the *names* of object-level formulae. As we are using MML as its own meta-language, this domain includes the names of *all* MML formulae. We use a *typed representation* (Hill and Lloyd, 1988) for the meta-language, and partition variables into two sorts, o, representing object-level terms, and μ, representing meta-level terms. The terms of MML can thus be partitioned into two sets:

$\mathcal{L}_{ot}$ — the set of *object terms*.
This contains those finite terms that are either variables of sort o or constants[3], or are generated by application of function symbols to other elements of $\mathcal{L}_{ot}$. Such terms have sort o and can be considered as the basic (object-level) terms of the language.

$\mathcal{L}_{\mu t}$ — the set of *meta-level terms*.
This set contains variables of sort μ, all the finite formulae constructed from elements of $\mathcal{L}_{\mu t}$, and predicates applied to terms in $\mathcal{L}_{\mu t} \cup \mathcal{L}_{ot}$. Such terms have sort μ and can be considered as the names of all the formulae in the language (i.e. the meta-level terms).

Thus, $\mathcal{L}_{\mu t}$ is effectively the set of well-formed formulae (wff) of the both the object and meta-levels, together with meta-level variables.

[3] All constants are object-level terms.

As an example, consider the term $p(f(a, b, c))$ where p is a predicate, f is a function and a, b and c are constants. Here, $f(a, b, c)$ is a term in $\mathcal{L}_{ot}$, yet when a predicate is applied to it, it becomes a term in $\mathcal{L}_{\mu t}$, i.e. $p(f(a, b, c))$ is a term in $\mathcal{L}_{\mu t}$.

Note that when we define WFF_m (well-formed formulae of MML) then for every formula in WFF_m, there is a corresponding term in $\mathcal{L}_{\mu t}$. For example, $p(f(a, b, c))$ is in WFF_m, but it is also a term in $\mathcal{L}_{\mu t}$. This duplication is essential as the elements of $\mathcal{L}_{\mu t}$ act as 'names' for the formulae in WFF_m. This will not cause any technical problems since elements of $\mathcal{L}_{\mu t}$ are only used as names for formulae. The construction of $\mathcal{L}_{\mu t}$ is a recursive (and long) process.

We now turn to the semantics of MML. Recall that the domain over which terms range is not simply the domain of $\mathcal{L}_{ot}$, it is also the domain of $\mathcal{L}_{\mu t}$, the names of formulae in MML. Note that, as we are using a temporal logic with non-rigid interpretations, we require that interpretations be indexed by the state (or world) at which the symbols are to be evaluated. As all variables have a sort, we can simply bind them to elements of the appropriate domain. In classical logics, the interpretation of a predicate symbol usually gives a signature of the form

$$\mathcal{D}^n \rightarrow \{T, F\}.$$

Thus, depending on the values of its arguments, the predicate is interpreted as either T or F. In MML, we not only have to index this interpretation with the state at which the predicate is to be evaluated, but there is also the possibility of the predicates' arguments being in $\mathcal{L}_{\mu t}$ (as well as $\mathcal{L}_{ot}$). Thus, we interpret predicates using

$$(\mathcal{S} \times (\mathcal{D} \cup \mathcal{T})^n) \rightarrow \{T, F\}$$

where $\mathcal{S}$ is the set of states (possible-worlds) and $\mathcal{T}$ is the domain associated with $\mathcal{L}_{\mu t}$, i.e. the ground terms in $\mathcal{L}_{\mu t}$.

One peculiarity of this approach is that if we have a variable assignment that maps x to φ, where x is of sort μ and $\varphi \in \mathcal{L}_{\mu t}$, then x is replaced by φ throughout the formula. Thus,

$$p(x) \wedge q(x, r(x))$$

becomes

$$p(\varphi) \wedge q(\varphi, r(\varphi)).$$

In summary, the main difference between MML and a standard first-order temporal logic (apart from the scoping of variable quantification, as described above) is that the interpretation of predicate symbols is not just a map from elements of the base domain to $\{T, F\}$, but is a map from elements of the base domain and the domain of names of formulae ($\mathcal{T}$) to $\{T, F\}$. (This is why, in the example given in appendix A.3, the interpretation function for predicate symbols maps formulae, as well as terms, to $\{T, F\}$.)

One further complication is that as we are dealing with a temporal logic, π_p is also parameterised by the state at which it is evaluated, i.e.,

$$\pi_p \colon (S \times \mathcal{L}_p) \rightarrow ((\mathcal{D} \cup \mathcal{T})^n \rightarrow \{T, F\}).$$

For an example of the use of this semantics, see appendix A.3.

4 Applications

We now give some examples of the use of meta-level representation in METATEM. The notation used will be that of MML and the underlying execution mechanism will be that outlined in section 1.

Some of the obvious applications of meta-level reasoning will be described, such as control of execution and the construction of specialized meta-interpreters, such as those for exploring possible futures. Apart from the applications described here, other aspects of meta-programming that are being explored elsewhere include the implementation of concurrent and object-oriented versions of METATEM, dynamic learning of temporal rules, and the development of compilers, debuggers and partial evaluators.

4.1 Building an interpreter for PML

To show how the meta-level techniques described above can be used in practice, we give, as an example, an interpreter for propositional METATEM (PML) written in MML. Note that the execution mechanism assumes that the MML interpreter contains a forward chaining inference mechanism, not only for temporal rules, but also for solving state-based constraints and that the variables within the rules are universally quantified.

To show the split of rules between state-based and temporal constraints, we will present these two categories separately. However, in a 'real' implementation, there may be no need for such a split.

First, we give the rules for a naive PML interpreter; later we add more sophisticated features. The *State Rules* for the interpreter (which is encapsulated within the `execute()` predicate[4]) are given in figure 1.

$$
\begin{array}{rcl}
\texttt{execute}(\Box\varphi) & \Leftrightarrow & \texttt{execute}(\varphi \wedge \bigcirc\Box\varphi) \\
\texttt{execute}(\Diamond\varphi) & \Leftrightarrow & \texttt{execute}(\varphi \vee (\neg\varphi \wedge \bigcirc\Diamond\varphi)) \\
\texttt{execute}(\neg p) & \Leftrightarrow & \neg\texttt{execute}(p) \\
\texttt{execute}(\varphi \wedge \psi) & \Leftrightarrow & \texttt{execute}(\varphi) \wedge \texttt{execute}(\psi) \\
\texttt{execute}(\varphi \Rightarrow \psi) & \Leftrightarrow & \texttt{execute}(\varphi) \Rightarrow \texttt{execute}(\psi) \\
\texttt{execute}(\varphi \vee \psi) & \Leftrightarrow & \texttt{execute}(\varphi) \vee \texttt{execute}(\psi) \\
(\texttt{prop}(p) \wedge \texttt{env}(p)) & \Rightarrow & (\texttt{execute}(p) \Leftrightarrow \texttt{read}(p)) \\
(\texttt{prop}(p) \wedge \texttt{comp}(p)) & \Rightarrow & (\texttt{execute}(p) \Leftrightarrow \texttt{do}(p))
\end{array}
$$

Figure 1: State Rules for a PML Interpreter

The basic *Temporal Rule* is

$$\bullet\texttt{execute}(\bigcirc\varphi) \Rightarrow \texttt{execute}(\varphi).$$

The idea behind these rules is that they are applied globally, with the translation rules being used to execute any current-state constraints, and the temporal rule being used to move from one state to the next. The predicates `prop()`, `env()`, `comp()`, `read()`, and `do()` are basic system predicates supplied by the base interpreter. `prop()` is true if its argument is a proposition, `env()` is true if its argument is controlled by the environment, `comp()` is true if its argument is controlled by this component, `read()` is true if its argument has been set to **true** by the environment, and `do()` is true when its argument is forced to be **true** by this component.

As an example of the execution of this meta-interpreter, consider the execution of the PML formula $r \Rightarrow \bigcirc\Diamond a$, where r is an environment proposition and a is a component proposition. Thus, we wish to make $\texttt{execute}(r \Rightarrow \bigcirc\Diamond a)$ true. The translation rules above are used to rewrite this formula as follows

* $\texttt{execute}(r \Rightarrow \bigcirc\Diamond a)$ becomes

  $$\texttt{execute}(r) \Rightarrow \texttt{execute}(\bigcirc\Diamond a)$$

  using the rule for '$\Rightarrow$',
* the object-level system interprets

  $$\texttt{execute}(r) \Rightarrow \texttt{execute}(\bigcirc\Diamond a)$$

  as

  'if $\texttt{execute}(r)$ then $\texttt{execute}(\bigcirc\Diamond a)$',
* as r is an environment proposition, $\texttt{execute}(r)$ becomes $\texttt{read}(r)$,
* if $\texttt{read}(r)$ is false, then the original formula is trivially satisfied, but if $\texttt{read}(r)$ is true, then $\texttt{execute}(\bigcirc\Diamond a)$ is not expanded further until the object-level interpreter moves to the next moment in time where the temporal rule is invoked, generating $\texttt{execute}(\Diamond a)$.

And so on. The computation continues forever unless the formulae being executed are contradictory. Note that the rewriting performed in every state will always terminate as we only use finite formulae and that once this rewriting has terminated, the object-level interpreter will move on to the next moment in time.

Now, notice that, given the rule for executing $\Diamond\varphi$ above, it is perfectly acceptable for the interpreter to continually execute $\neg\varphi$. We can try to avoid this behaviour and force the meta-interpreter to execute eventualities by adding the temporal rule

$$\bullet(\texttt{execute}(\Diamond\varphi) \wedge \neg\texttt{execute}(\varphi)) \Rightarrow \Diamond\texttt{execute}(\varphi)$$

Here the responsibility for executing the eventuality is passed on to the interpreter that is actually executing the meta-interpreter (i.e., the object-level interpreter). However, if the object-level interpreter does not guarantee to execute the eventuality, we are left with the same problem. A solution to this problem is to ensure that some form of loop-checking is present.

[4]Note that when we use equivalences such as $A \Leftrightarrow B$, where A is a past-time formula, we use this as an abbreviation for the rules $A \Rightarrow B$ and $\neg A \Rightarrow \neg B$. If we use $A \Leftrightarrow B$, where neither A nor B are past-time formulae, we use this as an abbreviation for $\bullet\,\mathbf{true} \Rightarrow ((A \wedge B) \vee (\neg A \wedge \neg B))$.

4.1.1 Loop Checking

A possible development of the interpreter is to incorporate heuristics for identifying potential looping in an execution of an object-level specification. For example, if we are operating without an environment, we can consider that if the set of commitments remains constant over a period of time, and that this set of commitments includes eventuality commitments then we are failing to satisfy these eventualities. So if we take the simple interpreter, defined above, we can amend the definition of $\texttt{execute}(\varphi \vee \psi)$ so that if one of the disjuncts represents an eventuality that has been outstanding for a sufficiently long time, then that disjunct will be chosen. The extra rule are given in figure 2.

$$\neg(\texttt{execute}(\varphi \vee \psi) \wedge \texttt{loop}(\varphi) \wedge \texttt{loop}(\psi))$$
$$(\texttt{execute}(\varphi \vee \psi) \wedge \neg\texttt{loop}(\varphi) \wedge \texttt{loop}(\psi)) \Rightarrow \texttt{execute}(\psi)$$
$$(\texttt{execute}(\varphi \vee \psi) \wedge \texttt{loop}(\varphi) \wedge \neg\texttt{loop}(\psi)) \Rightarrow \texttt{execute}(\varphi)$$
$$(\texttt{execute}(\varphi \vee \psi) \wedge \neg\texttt{loop}(\varphi) \wedge \neg\texttt{loop}(\psi)) \Rightarrow \texttt{execute}(\varphi) \vee \texttt{execute}(\psi)$$

Figure 2: Extra Rules for Loop-Checking

Here, the predicate `loop()` is defined so that it is true if the formula given as its argument has appeared in the set of commitments for the last n states (i.e. its argument has been an argument to `execute()` for the last n states, but is an eventuality that has not been satisfied in this time). If the specification is being executed without interacting with its environment, the value n can be extracted from the specification itself as the upper bound on the size of the automaton representing models of the specification (Vardi and Wolper, 1986)[5]. If interaction with the environment occurs, the value n cannot be deduced directly and must either be defined explicitly or derived through the use of heuristics.

4.2 Planning Linear Execution using a Branching Interpreter

Another example of the use of meta-level techniques in constraining the execution of METATEM is the use of a 'branching interpreter' to explore future paths. The idea here is that at certain stages in the execution process, when a choice is faced, a branching interpreter, represented by the predicate `branch-exec()`, is invoked to explore the future for a certain distance and to aid in the decision of what choice to make in the current state. Thus, most of the rules for `branch-exec()` are similar to those for `execute()`; some of the ones that are different are given in figure 3.

The first two rules show that when `branch-exec()` executes a choice, both choices are independently executed. A list of choices taken is recorded in the second argument of `branch-exec()`, and a measure of the "goodness" of the represented path is delivered in the third argument of `branch-exec()`. The final argument is a bound on the depth of the search.

$$\begin{aligned}&\texttt{branch-exec}(\varphi \vee \psi,\ [\varphi]\hat{}L,\ V_L,\ b)\\ &\quad\Rightarrow \texttt{branch-exec}(\varphi,\ L,\ V_L,\ b) \wedge \texttt{branch-exec}(\psi,\ R,\ V_R,\ b) \wedge \texttt{better}(V_L,\ V_R)\\ &\texttt{branch-exec}(\varphi \vee \psi,\ [\psi]\hat{}R,\ V_R,\ b)\\ &\quad\Rightarrow \texttt{branch-exec}(\varphi,\ L,\ V_L,\ b) \wedge \texttt{branch-exec}(\psi,\ R,\ V_R,\ b) \wedge \texttt{better}(V_R,\ V_L)\\ &\texttt{branch-exec}(\varphi,\ [\,],\ V,\ 0) \Leftrightarrow \texttt{value}(\varphi,\ V)\end{aligned}$$

$$\begin{aligned}&(\texttt{branch-exec}(\bigcirc\varphi,\ L,\ V,\ b) \wedge (b > 0))\\ &\quad\Rightarrow \texttt{branch-exec}(\varphi,\ L,\ V,\ b-1)\end{aligned}$$

Figure 3: Extra Rules for a Branching Interpreter

Thus, during execution, the base interpreter might choose to invoke `branch-exec()` at some stage to explore the future. Once the future has been simulated for a certain number of states, the values of the `branch-exec()` predicate will represent the possible futures up to that number of states. The base interpreter then chooses which of these possible futures gives the 'best' execution and then follows the plan represented by the argument L generated for that particular future. For example, the execution mechanism given earlier using `execute()` might be modified to incorporate the following rules[6]

$$\begin{aligned}&\texttt{execute}(\varphi \vee \psi)\\ &\quad\Rightarrow \texttt{branch-exec}(\varphi \vee \psi,\ L,\ V,\ 8) \wedge \texttt{planned-exec}(\varphi \vee \psi,\ L)\end{aligned}$$

Thus, the execution would follow the path represented in L and would be guided down this path by `planned-exec()`. The rules for `planned-exec()` would again be similar to those for `execute()`, the difference being in the rules representing guidance, such as

$$\texttt{planned-exec}(\varphi \vee \psi,\ [\varphi]\hat{}L) \Rightarrow \texttt{planned-exec}(\varphi,\ L)$$

When the path has been followed as far as possible, execution would revert back to being carried out by `execute()`, i.e.,

$$\texttt{planned-exec}(\varphi,\ [\,]) \Rightarrow \texttt{execute}(\varphi)$$

Thus, heuristics define the choice of which future is 'best'. For example, it may be that the 'best' future is the one with the least number of outstanding constraints.

4.3 A Simple Form of Planning

We can develop a system that incorporates a simple form of planning by utilising the following observations about METATEM.

[5]Note that this is only possible in a propositional language.

[6]Here, the branching interpreter searches forward for at most 8 states.

- In attempting to execute an eventuality, such as $\Diamond p$, the execution mechanism carries out goal-directed forward reasoning (where p represents the goal).
- The meta-level constraints on the execution mechanism effectively prune the search space and direct the execution towards the satisfaction of the eventualities.

Thus, we can implement a naive planning system by

1. representing the basic plan components as facts that are known at the beginning of the execution, and
2. representing schemas for combining separate subplans, to produce larger plans, as rules (or meta-rules) in the METATEM program.

Given a goal state, characterised by the formula φ, we then try to execute $\Diamond\varphi$. Given the basic execution mechanism, the system will search forward, by constructing linear paths, in an attempt to reach the goal. A more sophisticated system would incorporate the 'branching look-ahead' described above. This would generate a branching structure, representing prefices of all the possible paths that are being explored, and use this structure in deciding which path to follow.

Meta-level constraints can also be added to both prune the search space and direct the execution mechanism towards the goal.

Acknowledgements

This work was supported by ESPRIT under Basic Research Action 3096 (SPEC).

References

Abadi, M. (1987). *Temporal-Logic Theorem Proving*. PhD thesis, Department of Computer Science, Stanford University.

Allen, J. F. and Hayes, P. J. (1985). A Common Sense Theory of Time. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 528–531, Los Angeles, California.

Barringer, H., Fisher, M., Gabbay, D., Gough, G., and Owens, R. (1989). METATEM: A Framework for Programming in Temporal Logic. In *REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (LNCS Volume 430)*, pages 94–129, Mook, Netherlands. Springer Verlag.

Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. (1987). LUSTRE: A Declarative Language for Programming Synchronous Systems. In *Proceedings of the Fourteenth ACM Symposium on the Principles of Programming Languages*, pages 178–188, Munich, West Germany.

Clarke, E. M. and Emerson, E. A. (1982). Using Branching Time Temporal Logic to Synthesise Synchronisation Skeletons. *Science of Computer Programming*, 2:241–266.

Fisher, M. (1989). Implementing a Prototype METATEM Interpreter. SPEC Project Report, Department of Computer Science, University of Manchester.

Fisher, M. (1990). Meta-Programming in METATEM. Temple group report (draft), Department of Computer Science, University of Manchester.

Fujita, M., Kono, S., Tanaka, T., and Moto-oka, T. (1986). Tokio: Logic Programming Language based on Temporal Logic and its compilation into Prolog. In *3rd International Conference on Logic Programming (LNCS Volume 225)*, pages 695–708, London. Springer-Verlag.

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

Hattori, T., Nakajima, R., Sakuragawa, T., Niide, N., and Takenaka, K. (1986). RACCO: a modal-logic programming language for writing models of real-time process-control systems. Technical report, Research Institute for Mathematical Sciences, Kyoto University.

Hill, P. M. and Lloyd, J. W. (1988). Analysis of Meta-Programs. In *Proceedings of the Workshop on Meta-Programming in Logic Programming*, pages 27–42, University of Bristol, U.K.

Moszkowski, B. (1986). *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, U.K.

Nilsson, U. and Małuszyński (1990). *Logic, Programming and Prolog*. John Wiley and Sons, Chichester, U.K.,.

Owens, R. P. (1990). Using TEQUEL. Internal Report, Department of Computing, Imperial College.

Perlis, D. (1985). Languages with Self Reference I: Foundations. *Artificial Intelligence*, 25:301–322.

Pnueli, A. (1977). The Temporal Logic of Programs. In *Proceedings of the Eighteenth Symposium on the Foundations of Computer Science*, Providence.

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

Torsun, I. S. and Manning, K. J. (1990). Execution and Application of Temporal Modal Logic. Internal Report, Department of Computing, University of Bradford, U.K.

van Harmelen, F. (1988). A Classification of Meta-Level Architectures. In *Proceedings of the Workshop on Meta-Programming in Logic Programming*, pages 81–94, University of Bristol, U.K.

Vardi, M. Y. and Wolper, P. (1986). Automata-theoretic Techniques for Modal Logics of Programs. *Journal of Computer and System Sciences*, 32(2):183–219.

A Definition of MML

A.1 Syntax of MML

The language MML consists of the following symbols

- A set, $\mathcal{L}_p$, of *predicate symbols*.
  Associated with each predicate symbol, p, is a mapping, p_m, from $\{1, \ldots, n\}$ (where n is a non-negative integer representing the *arity* of the predicate symbol) to the set of sorts, $\{o, \mu\}$.

  Predicates with arity 0 are called propositions.

  The set, $\mathcal{L}_p$, is the disjoint union of the sets $\mathcal{L}_{pc}$ and $\mathcal{L}_{pe}$, of *component* and *environment* predicate symbols.

- A set, $\mathcal{L}_v$, of *variable symbols*.
  Associated with each variable symbol, v, is an element of the set of sorts, $\{o, \mu\}$, given by *sort(v)*.

- A set, $\mathcal{L}_c$, of *constant symbols*[7].
  The *sort* of any element of $\mathcal{L}_c$ is o.

- A set, $\mathcal{L}_f$, of *function symbols*.
  Associated with each function symbol, f, is an arity, given by *arity(f)*.

- Quantifiers $\exists_s$, $\forall_s$, where s is an element of the set of sorts, $\{o, \mu\}$, i.e., $\exists_o$, $\exists_\mu$, $\forall_o$, $\forall_\mu$.

- A set of *propositional connectives*, $\neg$, $\wedge$, $\vee$, $\Rightarrow$.

- A set of *temporal connectives*, $\bullet$, $\circledcirc$, $\mathcal{S}$, $\mathcal{Z}$, $\bigcirc$, $\odot$, $\mathcal{U}$, $\mathcal{W}$, $\diamondsuit$, $\square$.

As there are two disjoint sets of terms, variables are categorised into two sorts, o, representing object-level terms, and μ, representing meta-level terms.

The set of *object terms*, $\mathcal{L}_{ot}$, contains those terms that are either constants or variables of sort o, or are generated by application of function symbols to other elements of $\mathcal{L}_{ot}$. Such terms have sort o and can be considered as the basic terms of the language. $\mathcal{L}_{ot}$ is defined as follows.

1. Any constant, c, is in $\mathcal{L}_{ot}$.
2. Any variable, v, where $sort(v) = o$, is in $\mathcal{L}_{ot}$.
3. If $o_1, \ldots, o_n$ are in $\mathcal{L}_{ot}$, and f is a function symbol of arity $n > 0$, then $f(o_1, \ldots, o_n)$ is in $\mathcal{L}_{ot}$.

The set of *meta-level terms*, $\mathcal{L}_{\mu t}$, whose elements have sort μ, contain variables of sort μ, all the formulae constructed from elements of $\mathcal{L}_{\mu t}$, and predicates applied to terms in $\mathcal{L}_{\mu t} \cup \mathcal{L}_{ot}$.

$\mathcal{L}_{\mu t}$ is constructed as follows

1. Any variable, v, where $sort(v) = \mu$, is in $\mathcal{L}_{\mu t}$.

[7]Only object level constants and functions that apply to object level terms are allowed.

2. If φ and ψ are in $\mathcal{L}_{\mu t}$, then so are

$$\begin{array}{llll}
\textbf{true} & \textbf{false} & (\varphi) & \neg\varphi \\
\varphi \wedge \psi & \varphi \vee \psi & \varphi \Rightarrow \psi & \\
\bullet\varphi & \circledcirc\varphi & \varphi\,\mathcal{S}\,\psi & \varphi\,\mathcal{Z}\,\psi \\
\bigcirc\varphi & \odot\varphi & \varphi\,\mathcal{U}\,\psi & \varphi\,\mathcal{W}\,\psi \\
\square\varphi & \Diamond\varphi & &
\end{array}$$

3. If $x_1, \ldots, x_n$ are in $\mathcal{L}_{ot} \cup \mathcal{L}_{\mu t}$, and p is a predicate symbol of arity $n > 0$, and for each i from 1 to n, $p_m(i) = sort(x_i)$ then $p(x_1, \ldots, x_n)$ is in $\mathcal{L}_{\mu t}$.
   Note that any proposition symbol is in $\mathcal{L}_{\mu t}$.

Thus, $\mathcal{L}_{\mu t}$ is effectively the set of well-formed formulae (wff) for the object-level, together with meta-level variables.

A term is called *ground* if it contains no variable symbols. Let $\mathcal{T}$ be the subset of $\mathcal{L}_{\mu t}$ containing only ground variables[8].

The set of well-formed formulae of MML, WFF_m, is defined as follows.

1. If $x_1, \ldots, x_n$ are in $\mathcal{L}_{ot} \cup \mathcal{L}_{\mu t}$, and p is a predicate symbol of arity $n > 0$, and for each i from 1 to n, $p_m(i) = sort(x_i)$ then $p(x_1, \ldots, x_n)$ is in WFF_m.
   Thus, any proposition symbol is in WFF_m.
2. If φ and ψ are in WFF_m, then so are

$$\begin{array}{llll}
\textbf{true} & \textbf{false} & (\varphi) & \neg\varphi \\
\varphi \wedge \psi & \varphi \vee \psi & \varphi \Rightarrow \psi & \\
\bullet\varphi & \circledcirc\varphi & \varphi\,\mathcal{S}\,\psi & \varphi\,\mathcal{Z}\,\psi \\
\bigcirc\varphi & \odot\varphi & \varphi\,\mathcal{U}\,\psi & \varphi\,\mathcal{W}\,\psi \\
\square\varphi & \Diamond\varphi & &
\end{array}$$

3. If φ is in WFF_m, v is in $\mathcal{L}_v$, and *sort(v)=s*, then $\exists_s v.\ \varphi$ and $\forall_s v.\ \varphi$ are in WFF_m.

A.2 Semantics of MML

Recall that the domain over which terms range is not simply the domain of $\mathcal{L}_{ot}$ — terms also range over the domain of $\mathcal{L}_{\mu t}$ (the names of formulae in MML). Thus, there are two domains: the first is simply the 'real' object-level domain, to which elements of $\mathcal{L}_{ot}$ map; the other is the set of names of ground formulae in MML — these are what elements of $\mathcal{L}_{\mu t}$ map to.

The only difficulty here is what to do about variables and predicates. As all variables have a sort, we can bind them to elements of the appropriate domain. The interpretation of a predicate symbol usually gives a signature something like

$$\mathcal{D}^n \rightarrow \{T, F\}.$$

Thus, depending on the values of its arguments, the predicate is interpreted as either T or F. In MML, we also have the possibility of arguments being in $\mathcal{L}_{\mu t}$ (as well as $\mathcal{L}_{ot}$), so we interpret predicates as

$$(\mathcal{D} \cup \mathcal{T})^n \rightarrow \{T, F\}$$

where $\mathcal{T}$ is the domain associated with $\mathcal{L}_{\mu t}$.

Well-formed formulae of MML are interpreted over model structures of the form

$$\mathcal{M} = \langle S, \mathcal{R}, \mathcal{D}, \mathcal{T}, \pi_c, \pi_f, \pi_p \rangle$$

where

- S is a set of states,
- $\mathcal{R}$ is a relation over S,
- $\mathcal{D}$ is the object-level domain,
- $\mathcal{T}$ is the meta-level domain (a ground version of $\mathcal{L}_{\mu t}$),
- π_c is a map from $\mathcal{L}_c$ to $\mathcal{D}$,
- π_f is a map from $\mathcal{L}_f$ to $\mathcal{D}^n \rightarrow \mathcal{D}$, where n is the arity of f, and,
- π_p is a map from $S \times \mathcal{L}_p$ to $(\mathcal{D} \cup \mathcal{T})^n \rightarrow \{T, F\}$.

A *variable assignment* is a mapping from $\mathcal{L}_v$ to elements of $\mathcal{D} \cup \mathcal{T}$.

Given a variable assignment, V, and the valuation functions, π_c and π_f, associated with a particular model structure, a term assignment $\tau_{v\pi}$ is a mapping from $\mathcal{L}_{ot} \cup \mathcal{L}_{\mu t}$ to $\mathcal{D} \cup \mathcal{T}$ and is defined as follows.

if $c \in \mathcal{L}_c$, $\quad \tau_{v\pi}(c) = \pi_c(c)$

if $f \in \mathcal{L}_f$, and the *arity* of f is n,
$\quad \tau_{v\pi}(f(t_1, \ldots, t_n) = \pi_f(f)(\tau_{v\pi}(t_1), \ldots, \tau_{v\pi}(t_n))$

if $v \in \mathcal{L}_v$, $\quad \tau_{v\pi}(c) = V(c)$

if $p \in \mathcal{L}_p$, and the *arity* of p is n,
$\quad \tau_{v\pi}(p(t_1, \ldots, t_n) = p(\tau_{v\pi}(t_1), \ldots, \tau_{v\pi}(t_n))$

if $t = \textbf{true}$ or $t = \textbf{false}$, $\quad \tau_{v\pi}(t) = t$

if $t = \text{OP}\,t'$, where $\text{OP}, \in \{\neg, \bigcirc, \odot, \bullet, \circledcirc, \Diamond, \square\}$,
$\quad \tau_{v\pi}(t) = \text{OP}\ \tau_{v\pi}(t')$

if $t = t'\,\text{OP}\,t''$, where $\text{OP} \in \{\wedge, \vee, \Rightarrow, \mathcal{S}, \mathcal{Z}, \mathcal{U}, \mathcal{W}\}$,
$\quad \tau_{v\pi}(t) = \tau_{v\pi}(t')\ \text{OP}\ \tau_{v\pi}(t'')$

Note that no checking of the sorts of variables is carried out here; this is taken care of in the semantics of quantification.

The semantics of a well-formed MML formula is given with respect to a model structure, a state at which the formula is to be interpreted, and a variable assignment. The satisfaction relation, $\models$, relates such tuples to formulae of WFF_m as follows.

[8] For a more expressive meta-language this restriction that the elements of $\mathcal{T}$ must be grounded can be removed.

$\langle \mathcal{M}, s, V \rangle \models \neg\varphi$ iff not $\langle \mathcal{M}, s, V \rangle \models \varphi$

$\langle \mathcal{M}, s, V \rangle \models \varphi \vee \psi$ iff
$\langle \mathcal{M}, s, V \rangle \models \varphi$ or $\langle \mathcal{M}, s, V \rangle \models \psi$

$\langle \mathcal{M}, s, V \rangle \models \varphi \wedge \psi$ iff
$\langle \mathcal{M}, s, V \rangle \models \varphi$ and $\langle \mathcal{M}, s, V \rangle \models \psi$

ETC....

$\langle \mathcal{M}, s, V \rangle \models \forall_o x.\ \varphi$ iff
for all $d \in \mathcal{D}$,
$\langle \mathcal{M}, s, V \dagger [x \mapsto d] \rangle \models \varphi$

$\langle \mathcal{M}, s, V \rangle \models \forall_\mu x.\ \varphi$ iff
for all $t \in \mathcal{T}$,
$\langle \mathcal{M}, s, V \dagger [x \mapsto t] \rangle \models \varphi$

$\langle \mathcal{M}, s, V \rangle \models \exists_o x.\ \varphi$ iff
there exists $d \in \mathcal{D}$ such that
$\langle \mathcal{M}, s, V \dagger [x \mapsto d] \rangle \models \varphi$

$\langle \mathcal{M}, s, V \rangle \models \exists_\mu x.\ \varphi$ iff
there exists $t \in \mathcal{T}$ such that
$\langle \mathcal{M}, s, V \dagger [x \mapsto t] \rangle \models \varphi$

$\langle \mathcal{M}, s, V \rangle \models p(x_1, x_2, \ldots, x_n)$ iff
$\pi_p(s, p)(\tau_{v\pi}(x_1), \tau_{v\pi}(x_2), \ldots, \tau_{v\pi}(x_n)) = T$

We will usually deal with closed formulae, i.e., formulae containing no free variables. In this case, we use the empty mapping, [ ], as the initial variable assignment.

In summary, the main difference between MML and standard temporal logics is that the interpretation of predicate symbols is not just a map from element of the base domain to $\{T, F\}$, but is a map from elements of either the base domain or the domain of names of formulae ($\mathcal{T}$) to $\{T, F\}$. (This is why, in the example that follows, π_p maps formulae, as well as terms, to $\{T, F\}$.)

A.3 Example

We now give an example of a model for an MML formula and show how the semantics can be used to show that the formula is satisfied in the model. This example is relevant to the applications described in section 4.

Consider the model

$$D = \{\},$$

$$T = \{r,\ q,\ \bigcirc r,\ etc\ \},$$

$$\pi_c = [\,],$$

$$\pi_f = [\,],$$

$$\begin{aligned}\pi_p = \ & [0 \mapsto \{\texttt{execute}(\bigcirc r),\ \texttt{execute}(\bigcirc q)\}] \\ & [1 \mapsto \{\texttt{execute}(r),\ \texttt{execute}(\bigcirc q),\ \texttt{execute}(q)\}] \\ & \ldots etc\ldots.\end{aligned}$$

Here, we use an abbreviated representation of π_p, where each state maps to the predicate applications that are true in that state. We now consider the satisfaction of the following formula in the above model when interpreted at state 1.

$$\forall_\mu x.\ \bullet\texttt{execute}(\bigcirc x) \Rightarrow \texttt{execute}(x)$$

The model satisfies the formula if

$$\langle \mathcal{M}, 1, [\,] \rangle \models \forall_\mu x.\ \bullet\texttt{execute}(\bigcirc x) \Rightarrow \texttt{execute}(x).$$

This is satisfied if for all t in $\mathcal{T}$,

$$\langle \mathcal{M}, 1, [x \mapsto t] \rangle \models \bullet\texttt{execute}(\bigcirc x) \Rightarrow \texttt{execute}(x).$$

So, for all t in $\mathcal{T}$,

if

$$\langle \mathcal{M}, 1, [x \mapsto t] \rangle \models \bullet\texttt{execute}(\bigcirc x)$$

then

$$\langle \mathcal{M}, 1, [x \mapsto t] \rangle \models \texttt{execute}(x).$$

Assuming that $0\mathcal{N}1$, this reduces to,

if

$$\langle \mathcal{M}, 0, [x \mapsto t] \rangle \models \texttt{execute}(\bigcirc x)$$

then

$$\langle \mathcal{M}, 1, [x \mapsto t] \rangle \models \texttt{execute}(x),$$

and finally to,

if

$$\pi_p(0, \texttt{execute})(\bigcirc t) = T$$

then

$$\pi_p(1, \texttt{execute})(t) = T.$$

Now, in state 0, $\texttt{execute}(\bigcirc t)$ is true for $t \in \{r,\ q\}$. Thus we must show that in state 1, both $\texttt{execute}(r)$ and $\texttt{execute}(q)$ are true. As this is so, the model satisfies the formula (at least when interpreted at state 1).

Further examples and details of both MML and FML* are given in (Fisher, 1990).