Department of Computer Science
University of Manchester

Manchester M13 9PL, England

Technical Report Series

UMCS–92–2–1

Michael Fisher and Philippe Noël

Transformation and Synthesis in METATEM
Part I: Propositional METATEM

Transformation and Synthesis in METATEM Part I: Propositional METATEM*

Michael Fisher and Philippe Noël

The TEMPLE Group
Department of Computer Science
University of Manchester
Manchester U.K.

{michael,philippe}@cs.man.ac.uk ...ukc!man.cs!{michael,philippe}

Abstract

In this report, the syntactic manipulation of temporal logic programs is considered. Transformation rules are provided for a temporal programming language that forms part of the METATEM framework for executable temporal logics [2]. Soundness of the various transformations is shown and several applications are given, such as the production of a normal form for programs, which is the basis of both compilation and resolution techniques. The use of transformation techniques in METATEM program synthesis is also considered. Here, a high-level specification, represented as a non-deterministic METATEM program is translated into an 'implementation', which is again represented as a METATEM program. The synthesised program is deterministic in that, for any action by the environment, the component has at least one possible (deterministic) response.

*This work was supported both by ESPRIT under Basic Research Action 3096 (SPEC), and by SERC under Research Grant GR/F 30123.

Contents

1 Introduction **2**

2 Temporal Logic **4**
2.1 Syntax 4
2.2 Semantics 5

3 METATEM **7**

4 Transformation Rules for METATEM Programs **9**
4.1 Eliminating Past-Time Operators 11
4.2 Eliminating Future-Time Operators 13
4.3 Manipulating METATEM Rules 13
4.4 Resolution Transformations 15

5 Correctness of the Transformations **17**
5.1 Correctness of Semantics Preserving Transformations 17
5.2 Correctness of Satisfiability Preserving Transformations 18
5.3 Correctness of Determinising Transformations 18

6 Applications of METATEM Program Transformation **20**
6.1 Transforming Programs into a Normal Form 20
6.2 Compilation of METATEM Programs 21
6.3 Temporal Resolution 21
6.4 Deriving Deterministic METATEM Programs 21
6.4.1 Automatic synthesis 21
6.4.2 Controlled synthesis 24

7 Conclusions and Future Work **25**

8 Acknowledgements **25**

References **26**

1 Introduction

Given a specification for a system, φ, there are several ways of deriving an implementation satisfying the specification. Three of the most widespread are:

1. Specification Refinement.
2. Program Transformation.
3. Automatic Program Synthesis.

The most extreme views are refinement and synthesis; transformation encompasses the variety of techniques in between (and including) these two.

The classical view of specification refinement is that at each refinement step, a new specification is developed and is shown, via some verification technique, to satisfy the specification at the previous level [13]. This process continues until the specification developed 'matches' some actual implementation. This 'matching' often involves the use of an appropriate semantics for the implementation language and a proof of containment between the behaviours of the specification and the behaviours of the implementation. In this approach, *any* refinement step can be generated, but such a refinement must be verified as being consistent with the previous specification. The specifier has complete control over the way in which the implementation is generated.

At the other extreme is automatic program synthesis. In this approach, an implementation is synthesised directly from a specification [5]. Thus, a 'compilation-like' mechanism is used to translate statements in the specification language directly into the implementation language. Here, no explicit verification is required as there are no explicit refinement operations. However, not only is program synthesis computationally expensive (and often intractable), but the program developer has no control over the form that the implementation takes. In particular, the synthesis mechanism can not easily be changed to generate alternate implementations, for example if certain forms of implementation run more efficiently on the particular target architecture.

Between the refinement and synthesis approaches lie a range of alternatives and *program transformations* can be used to encompass this range of development [6]. Program transformations represent a fixed set of rules for translating statements in the specification language. Since each of these rules is sound, no extra verification effort need be applied. With many specification systems, the set of commonly used transformations is often small. Thus, with a well chosen set of transformation rules, the specifier can transform the specification as required, without the extra effort of verifying any refinement. In some frameworks, the possible transformations are so rich that they can be combined to give a 'synthesis-like' procedure.

Though not restricted to executable specification languages, program transformations have been most widely applied in high-level functional and logic-based languages. For example, program transformation has been widely used as a mechanism for translating between recursive and iterative functional programs [4], and as a mechanism for improving the efficiency of logic programs [19].

Thus, as program transformation has been shown to be particularly applicable to high-level languages and executable specifications, it is natural to develop transformation techniques for METATEM, which fits into both of these categories. Though METATEM is a framework for programming in temporal logic, we will develop transformations for use in a particular instance of this framework, namely that language based on a propositional, discrete, linear temporal logic [2].

We describe and justify a variety of transformations for propositional METATEM programs and apply these transformations in several areas. We are particularly interested in applying transformations as part of a 'pre-compilation' phase and in order to synthesise deterministic programs from non-deterministic specifications. Though the language is propositional, and hence will not involve transformations analogous to *fold* and *unfold* [4, 19], the transformations produced are still quite complex as they need to be able to remove some of the temporal operators from programs[1] and to schedule events in time.

This report is structured as follows. In §2 and §3, an introduction to the temporal logic used, and to the METATEM style of executable logic, is given. In §4 various transformation rules are developed and categorised, with more advanced transformations being described in §4.4 and §6.4.

Several applications of transformation in METATEM will be discussed in §6. These include transforming METATEM programs into a normal form (§6.1) and the synthesis of deterministic METATEM programs (§6.4). Consistency proofs for various transformations are given in §5 and, finally, in §7, conclusions and future work in this area will be highlighted.

We begin by giving a brief overview of temporal logic in general, and the variety that we will use, in this paper, for our METATEM programs.

[1] Temporal operators represent restricted forms of quantification — thus propositional temporal logic is strictly more expressive than classical propositional logic.

2 Temporal Logic

Temporal logic can be seen as classical logic extended with various modalities. Commonly, these are $\Diamond$, $\Box$, and $\bigcirc$. The intuitive meaning of these connectives is as follows: $\Diamond A$ is true now if A is true *sometime* in the future; $\Box A$ is true now if A is true *always* in the future; and $\bigcirc A$ is true now if A is true at the *next* moment in time. Connectives analogous to the future-time operators are added to enable formulae to refer to the *past*, for example $\blacklozenge$ ('sometime in the past') and $\blacksquare$ ('always in the past'). As models of the logic have a unique start state, termed the *beginning of time*, two different last-time operators are used. These are the *strong* last-time operator, '$\circledcirc$', and the *weak* last-time operator, '$\bullet$'. The difference between these two is that for any formula A, $\circledcirc A$ is false at the beginning of time, while $\bullet A$ is true at the beginning of time.

In METATEM, the set of basic propositions of the underlying logic is partitioned between *environment* and *component* propositions. Intuitively, component propositions are those whose truth values are controlled by the particular METATEM component, while the truth values of environment propositions are controlled by some other agent in the component's environment. From a pragmatic point of view, this separation between a component specification and its environment is natural and has been widely used.

The temporal logic, based on discrete, linear models with finite past and infinite future, is called PML. This will provide the basis for the version of METATEM that we use in this report and the syntax and semantics of PML are outlined below.

2.1 Syntax

Well-formed formulae of PML (WFF) are generated from the set of connectives together with the set of propositions, $\mathcal{L}$, as follows. (Note that $\mathcal{L}$ is partitioned into sets of *component* and *environment* propositions, $\mathcal{L}_c$ and $\mathcal{L}_e$ respectively.)

- any element of $\mathcal{L}$ is in WFF,
- if A and B are in WFF, then the following are in WFF

$\neg A$	$A \vee B$	$A \wedge B$	$A \Rightarrow B$
$\Diamond A$	$\Box A$	$A \mathcal{U} B$	$A \mathcal{W} B$
$\blacklozenge A$	$\blacksquare A$	$A \mathcal{S} B$	$A \mathcal{Z} B$
$\bigcirc A$	$\circledcirc A$	$\bullet A$	(A)

Further sub-classifications of temporal formulae are defined as follows. A *literal* is defined as either a proposition (i.e., an element of $\mathcal{L}$), or the negation of a proposition.
A *State-formula* is either a literal or a boolean combination of other state-formulae.
Future-time formulae (non-strict) are defined as follows

- if A is a state-formula, then A is a future-time formula,
- if A and B are future-time formulae, then $\neg A$, $A \wedge B$, $A \vee B$, $A \Rightarrow B$, $A \mathcal{U} B$, $A \mathcal{W} B$, $\bigcirc A$, $\Diamond A$, and $\Box B$ are all future-time formulae.

Strict past-time formulae are defined as follows

- if A and B are either state-formulae or strict past-time formulae, then $\circledcirc A$, $\bullet B$, $A \mathcal{S} B$, $A \mathcal{Z} B$, $\blacklozenge A$, and $\blacksquare B$ are all strict past-time formulae,
- if A and B are strict past-time formulae, then $\neg A$, $A \wedge B$, $A \vee B$, and $A \Rightarrow B$ are all strict past-time formulae.

2.2 Semantics

The models of PML are discrete and linear model with finite past and infinite future, i.e.

$$\sigma = s_0, s_1, s_2, s_3, \ldots$$

Here, each s_i is called a *state* and is a subset of $\mathcal{L}$ representing the propositions that are true at the i^{th} moment in time. An interpretation for this logic is defined as a pair (σ, i), where σ is the model and i the index of the state at which the temporal statement is to be interpreted.

A semantics for well-formed temporal formulae is given as a relation between interpretations and formulae, and is defined inductively as follows, with the (infix) semantic relation being represented by $\models$. The semantics of a proposition is defined by the valuation given to it in the model:

$$(\sigma, i) \models p \quad \text{iff} \quad p \in s_i \qquad [\text{for } p \in \mathcal{L}].$$

The semantics of the standard propositional connectives are as in classical logic, for example

$$(\sigma, i) \models A \vee B \quad \text{iff} \quad (\sigma, i) \models A \quad \text{or} \quad (\sigma, i) \models B.$$

The semantics of the unary future-time temporal operators is defined as follows

$$\begin{array}{lll} (\sigma, i) \models \bigcirc A & \text{iff} & (\sigma, i+1) \models A \\ (\sigma, i) \models \Diamond A & \text{iff} & \text{there exists a } j \geq i \text{ such that } (\sigma, j) \models A \\ (\sigma, i) \models \Box A & \text{iff} & \text{for all } j \geq i \text{ then } (\sigma, j) \models A. \end{array}$$

The two binary future-time temporal operators are interpreted as follows

$$\begin{array}{lll} (\sigma, i) \models A \mathcal{U} B & \text{iff} & \text{there exists a } k \geq i \text{ such that } (\sigma, k) \models B \\ & & \text{and for all } i \leq j < k \text{ then } (\sigma, j) \models A \\ (\sigma, i) \models A \mathcal{W} B & \text{iff} & (\sigma, i) \models A \mathcal{U} B \quad \text{or} \\ & & \text{for all } j \geq i \text{ then } (\sigma, j) \models A \end{array}$$

If past-time temporal formulae are interpreted at a particular state, s_i, then states with indices less than i are *in the past* of the state s_i. The semantics of unary past-time operators are given as follows:

$$\begin{array}{l} (\sigma, i) \models \bullet A \text{ iff } i = 0 \quad \text{or} \quad (\sigma, i-1) \models A \\ (\sigma, i) \models \circledcirc A \text{ iff } i > 0 \quad \text{and} \quad (\sigma, i-1) \models A \\ (\sigma, i) \models \blacklozenge A \text{ iff there exists a } j \text{ such that } 0 \leq j < i \text{ and } (\sigma, j) \models A \\ (\sigma, i) \models \blacksquare A \text{ iff for all } j \text{ such that } 0 \leq j < i \text{ then } (\sigma, j) \models A. \end{array}$$

Note that, in contrast to the future-time operators, the $\blacklozenge$ ('sometime in the past') and $\blacksquare$ ('always in the past') operators are interpreted as being *strict*, i.e., the current index is not included in the definition. Also, as there is a unique start state, $\bullet$**false** is *only* true when interpreted at the beginning of time; otherwise it is false.

Apart from their strictness, binary past-time operators are similar to their future-time counterparts; their semantics is defined as follows.

$$\begin{array}{lll}
(\sigma, i) \models A \,\mathcal{S}\, B \text{ iff} & \text{there exists a } k \text{ such that } 0 \leq k < i \text{ and } (\sigma, k) \models B \text{ and} \\
& \text{for all } j \text{ such that } k < j < i \text{ then } (\sigma, j) \models A \\
(\sigma, i) \models A \,\mathcal{Z}\, B \text{ iff} & \text{for all } j \text{ such that } 0 \leq j < i \text{ then } (\sigma, j) \models A \quad \text{or} \\
& (\sigma, i) \models A \,\mathcal{S}\, B
\end{array}$$

The $\Diamond$ and $\Box$ (and their past-time counterparts) can be derived from the $\mathcal{U}$ and $\mathcal{W}$ operators ($\mathcal{S}$ and $\mathcal{Z}$) respectively as follows:

$$\begin{array}{rcl}
\Diamond A & \equiv & \textbf{true}\,\mathcal{U}\, A \\
\Box A & \equiv & A\,\mathcal{W}\,\textbf{false}
\end{array}$$

As an interpretation consists of a model/state-index pair, we say that a wff is satisfied in a particular model, at a particular state. The terminology is often extended to include a wff being *satisfied in a model*. A formula is satisfied in a model if it is satisfied in that model, at the beginning of time.

Finally, though the METATEM programs we consider do not contain any explicit quantification, we require a definition of quantification over propositions in order to prove the soundness of certain transformations. This quantification over propositions was developed by Wolper for QPTL [20].

We introduce the symbol '$\exists$', and extend the definition of well-formed formulae to include '$\exists p.\, A$', where p is a proposition and A is a well-formed formula. The semantics of such quantification is given using the *substitutional* interpretation [12]:

$$\begin{array}{ll}
(\sigma, i) \models \exists p.\, A \text{ iff} & \text{there exists a model } \sigma' \text{ such that } (\sigma', i) \models A \\
& \text{and } \sigma \text{ differs from } \sigma' \text{ in at most the valuation given to } p.
\end{array}$$

Universal quantification can be defined from existential quantification in the usual manner.

3 METATEM

METATEM [2] is a framework for executable temporal logics, within which programming languages based on a variety of temporal logics have been developed. The variety of METATEM considered in this report is that of PML executed using a particular mechanism.

A METATEM program consists of a set of *if-then* rules couched in terms of temporal logic. Each rule is of the form

$$P \;\Rightarrow\; F$$

where P refers only to properties of the past and present and F refers only to properties of the present and future. Thus, properties of the past trigger actions in, or constraints upon, the future. Each of these rules is applied at every moment in time.

Given a set of METATEM rules

$$\{P_i \Rightarrow F_i \mid 1 \leq i \leq n\}$$

comprising a METATEM program, the temporal formula associated with this program is

$$\Box \bigwedge_{i=1}^{n} (P_i \Rightarrow F_i)\,.$$

Note that, in this variety of METATEM, environment propositions are only allowed in the P_i component and, thus, the component is not allowed to 'overconstrain' its environment. Also, in PML, any arbitrary formula can be 'separated' into its past, present and future-time components [11].

Execution of temporal formulae within METATEM is carried out using the 'Imperative Future' paradigm [11, 2]. The basis of this approach is that if the past-time formulae in a rule are satisfied (declaratively) over the history of the component, then the future-time formulae associated with that rule must be satisfied (imperatively).

Example An example used extensively in papers on METATEM [2] is that of a resource controller. This is a component that accepts requests for a resource from several processes and allocates the resource to only one of these processes at each moment in time. A simple version of this object is given below.

$$\begin{aligned} \circledcirc\, r1 &\;\Rightarrow\; \Diamond a1 \\ \circledcirc\, r2 &\;\Rightarrow\; \Diamond a2 \\ \bullet\, \mathbf{true} &\;\Rightarrow\; \neg(a1 \wedge a2) \end{aligned}$$

Here, '*a1*' and '*a2*' are *component* propositions (i.e. their interpretation is controlled by the component) and '*r1*' and '*r2*' are *environment* propositions. The first rule states that if *r1* was satisfied in the last moment in time, then *a1* will eventually be made true; the second rule states a similar constraint between *r2* and *a2*. Thus, once *r1* has been set to true, the component must take steps to satisfy the imperative $\Diamond a1$, and similarly for *r2* and *a2*. The final rule implements mutual exclusion, stating that *a1* and *a2* must never be satisfied at the same moment in time[2].

[2]Note that here we will not consider the operational implementation of such an approach. An obvious way to look at such a system is as separate processes sharing certain variables. For a more detailed account of a concurrent operational model for METATEM, see [7].

Thus, as the rules are interpreted at every moment in time, a new set of constraints on the present and future execution of the program is generated at every state. The METATEM execution mechanism attempts to satisfy these constraints. For example, making *a2* true satisfies the constraint $\Diamond a2$. In fact, the execution mechanism does attempt to satisfy such eventualities as soon as possible. However, it may be the case that several of the constraints to be satisfied are eventualities containing incompatible formulae. For example, in the resource controller above, if $r1$ and $r2$ are true in the last moment in time, then the constraint set is $\{\Diamond a1,\ \Diamond a2\}$. Though the execution mechanism will attempt to satisfy both of these as soon as possible, it can only satisfy one at each moment in time. It is possible that this choice may lead to one of the eventualities being continually unsatisfied. To avoid this, the execution mechanism, when given a choice between satisfying several incompatible eventualities, chooses to satisfy one of the eventualities that have been unsatisfied for the longest time. This not only ensures that the execution mechanism is sound, but enforces a form of *fairness* in the execution of such constraints.

The style of executable logic represented by METATEM has several important advantages both over those languages based on classical logic and over those without any logical foundation. Temporal logic is a natural and powerful formalism in which the dynamic properties of systems can be expressed. Propositional temporal logic, with the standard temporal operators, is strictly more expressive than classical propositional logic. For example, the temporal operator '$\Diamond$', representing 'sometime in the future', effectively encodes a form of existential quantification over time. Thus $\Diamond p$, representing the statement 'p will be satisfied sometime in the future' cannot be expressed in a classical propositional logic. Though such statements can be expressed in classical first-order logic (for example, $\Diamond p$, can be represented as $\exists t.\ (t > now) \wedge sat(p, t)$, where $sat(p, t)$ is true if p is satisfied at time t), this encoding of temporal properties is unnatural and non-intuitive. Languages based on classical logics, such as Pure Prolog, suffer from such problems, when representing dynamic systems.

METATEM provides a rule-based formalism that is both natural (i.e. knowledge is represented in the 'past implies future' form) and flexible. Though obviously different, METATEM programs can express many of the same types of properties as systems based on production rules, particularly the representation of knowledge and the encoding of heuristics.

In theory, METATEM also supports meta-level reasoning. The logical basis of the framework includes the ability to name and manipulate formulae and rules and, thus, meta-level capabilities are provided as basic mechanisms within METATEM [3]. Such mechanisms provide a wide range of flexibility in developing, reconfiguring and dynamically modifying the behaviour of systems. Meta-level techniques also allow standard programming tools to be developed within a consistent logical framework. In METATEM family of logics, no explicit distinction between control information and data is made. Whether information is used as data or control depends entirely on whether it is used at the object or meta-levels. At present only the logical basis for this approach has been outlined and no implementation has yet been produced.

We now look in more detail at specific transformation rules for a propositional version of METATEM.

4 Transformation Rules for METATEM Programs

The transformation rules we introduce are of the form

$$\tau: 2^{\mathrm{WFF}} \longrightarrow 2^{\mathrm{WFF}}$$

Thus, a transformation translates a set of METATEM rules to another, possibly different, set of METATEM rules. Recall that, as sets of METATEM rules of the form $\{P_i \Rightarrow F_i \mid 1 \leq i \leq n\}$ represent the temporal formula

$$\Box \bigwedge_{i=1}^{n} (P_i \Rightarrow F_i)$$

then a transformation τ translating $\{P_i \Rightarrow F_i \mid 1 \leq i \leq n\}$ to $\{Q_j \Rightarrow G_j \mid 1 \leq j \leq m\}$, represents a translation from the formula above to

$$\Box \bigwedge_{j=1}^{m} (Q_j \Rightarrow G_j).$$

Before describing the particular transformation rules used in METATEM, we will introduce the following categorisations, which will be used to partition the transformation rules.

Definition 1 *A Semantics Preserving Transformation, τ, preserves the semantics of the formula that it is applied to, modulo any new propositions introduced. Thus, the models of an arbitrary formula, A, are the same as those of $\tau(A)$, apart from the valuation given to any new proposition symbol.*

The logical characterisation of these conditions is simply that $\vdash_{\mathrm{QPTL}} A \Leftrightarrow \exists x.\ \tau(A)$, where 'x' represents a proposition that does not occur in A.

Examples The following are both semantics preserving transformations (note that Ψ here represents a, possibly empty, set of METATEM rules).

- $\tau_{\wedge}(\{P \Rightarrow A \wedge B\} \cup \Psi) = \{P \Rightarrow A, P \Rightarrow B\} \cup \Psi$.
- $\tau_{\Box}(\{P \Rightarrow \Box B\} \cup \Psi) = \{P \Rightarrow (x \wedge B),\ \bigcirc x \Rightarrow (x \wedge B)\} \cup \Psi$.

  Here, 'x' is a *new* proposition symbol, i.e. it has not appeared in the program before. Given a function, *props*, which returns the set of proposition symbols used in a set of formulae, this condition can be represented by $x \notin \mathit{props}(\{P, B\} \cup \Psi)$.

Definition 2 *A Satisfiability Preserving Transformation, τ, ensures that for any arbitrary formula, A, if A is satisfiable, then $\tau(A)$ is satisfiable.*

Example A simple example of this is a transformation rule representing classical resolution:

$$\tau_R(\{P \Rightarrow A \vee b, Q \Rightarrow C \vee \neg b\} \cup \Psi) \;=\; \{(P \wedge Q) \Rightarrow (A \vee C)\} \cup \Psi$$

Definition 3 *A Determinising Transformation, τ, restricts the semantics of the formula, and does not necessarily preserve satisfiability. Thus, for any arbitrary formula, A, if a model, σ, satisfies $\tau(A)$, then it satisfies A.*

Examples The following are both determinising transformations.

- $\tau_\vee(\{P \Rightarrow A \vee B\} \cup \Psi) \;=\; \{P \Rightarrow A\} \cup \Psi$.
- $\tau_{\Diamond}(\{P \Rightarrow \Diamond A\} \cup \Psi) \;=\; \{P \Rightarrow A\} \cup \Psi$.

Such transformations are used as part of the synthesis procedure described in §6.4, though care is taken to apply them in such a way that satisfiability is not lost. For example, if we have the METATEM program rules

$$\begin{array}{lcl} \bullet\,\mathbf{false} & \Rightarrow & \neg l \\ \bullet\,\mathbf{false} & \Rightarrow & \Diamond l \end{array}$$

a blind application of $\tau_{\Diamond}$ would lead to the unsatisfiable set of rules

$$\begin{array}{lcl} \bullet\,\mathbf{false} & \Rightarrow & \neg l \\ \bullet\,\mathbf{false} & \Rightarrow & l \end{array}$$

Consequently, the transformations used in §6.4 are often combinations of determinising and satisfiability preserving transformations.

One example of determinising transformations is the actual *execution* of METATEM programs. This applies a series of determinising transformations to an original program producing, as a side effect, the sequence of states of the computation. Though satisfiability may be lost by such an application of determinising transformations, it is recovered through the use of backtracking. Thus, when an unsatisfiable set of formulae is produced, the transformations are undone and alternate ones applied.

We will now look at a variety of transformation rules for METATEM, categorising them according to the properties that they preserve. The rules will be given in the following style.

$$\begin{array}{lcl} P_1 & \Rightarrow & F_1 \\ \cdots & & \cdots \\ P_n & \Rightarrow & F_n \\ \hline Q_1 & \Rightarrow & G_1 \\ \cdots & & \cdots \\ Q_m & \Rightarrow & G_m \end{array}$$

Here, the set of METATEM rules that are transformed are placed above the line, while the set of METATEM rules generated are placed below. All the variables represent formulae unless otherwise stated. Often, new proposition symbols are introduced in the transformation rules. For simplicity, we will assume that all lower-case letters represent propositions and the symbols x, y, and z represent such new proposition symbols.

The definition of the transformations that follows involves specifying various logical formulae. Rather than explicitly enumerating all the varieties of formulae, logical contexts will be used. These contexts are represented by *context functions* such as $\mathcal{P}$, $\mathcal{F}$, and $\mathcal{L}$, which describe particular functions over well-formed formulae. For example, given a context function, $\mathcal{C}$, representing '$a \wedge$', this can be applied to the formula $e \vee f$ producing the formula $a \wedge (e \vee f)$, i.e.,

$$\mathcal{C}(e \vee f) \;=\; a \wedge (e \vee f).$$

The use of such contexts reduces the number of transformation rules required and generalises the transformations performed. Rather than using the specific context function given in the example above, three general (and arbitrary) context functions will be used:

$\mathcal{P}$ — an arbitrary (strict) past-time context function. Given a present or past-time formula r, $\mathcal{P}(r)$ is also past-time formula.

$\mathcal{F}$ — an arbitrary future-time context function. Thus, given a future-time formula, g, $\mathcal{F}(g)$ is also a future-time formula.

$\mathcal{L}$ — an arbitrary last-time context function, representing either of the last-time operators, $\bullet$ or $\circledcirc$. Thus, given a present or past-time formula, m, then $\mathcal{L}(m)$ is either $\bullet\, m$ or $\circledcirc\, m$.

Note that $\mathcal{F}$ can be the identity function, while $\mathcal{L}$ and $\mathcal{P}$ can not.

The first set of rules considered are used to eliminate various temporal operators from METATEM programs and are all semantics preserving transformations.

4.1 Eliminating Past-Time Operators

The rules given below can be used to reduce the past-time part of every rule in a METATEM program so that it contains only one last-time operator (i.e., $\bullet$ or $\circledcirc$). Certain operators ($\blacklozenge$ and $\blacksquare$) can be easily removed by rewriting them in terms of $\mathcal{S}$ and $\mathcal{Z}$ respectively:

$$\begin{aligned} \blacklozenge A &\equiv \mathbf{true}\, \mathcal{S}\, A \\ \blacksquare A &\equiv A\, \mathcal{Z}\, \mathbf{false} \end{aligned}$$

The rest of the past-time temporal operators require more complicated transformations. The first transformation removes the *since* operator, $\mathcal{S}$. Note that the rule $P \Leftrightarrow F$ is used as an abbreviation for the two rules $P \Rightarrow F$ and $(\neg P) \Rightarrow (\neg F)$.

$$\frac{\begin{aligned} \mathcal{P}(A\, \mathcal{S}\, B) &\Rightarrow F \\ \circledcirc (B \vee (A \wedge x)) &\Leftrightarrow x \end{aligned}}{\mathcal{P}(x) \Rightarrow F}$$

This transformation rule can be applied to any program rule of the form $\mathcal{P}(A\,\mathcal{S}\,B) \;\Rightarrow\; F$, where A and B are non-strict past-time formulae, F is a non-strict future-time formula and $\mathcal{P}$ is an arbitrary past-time context function. The transformation rule states that such a program rule can be replaced by the new rules $\circledcirc(B \vee (A \wedge x)) \;\Leftrightarrow\; x$ (which can in turn be expanded into two separate rules) and $\mathcal{P}(x) \;\Rightarrow\; F$, where x is a new proposition. Thus, for example, the program rule $\bullet(p\,\mathcal{S}\,q) \;\Rightarrow\; a \wedge \Diamond b$ can be transformed to

$$\begin{array}{rcl} \circledcirc(q \vee (p \wedge x)) & \Leftrightarrow & x \\ \bullet x & \Rightarrow & a \wedge \Diamond b \end{array}$$

In the case where $A\,\mathcal{S}\,B$ does not appear in a past-time context, we cannot simply replace $A\,\mathcal{S}\,B$ by x, as this would leave a proposition as the antecedent of a rule. Thus, the following transformation must be used.

$$\frac{\begin{array}{rcl} A\,\mathcal{S}\,B & \Rightarrow & F \end{array}}{\begin{array}{rcl} \circledcirc(B \vee (A \wedge x)) & \Leftrightarrow & x \\ \bullet\,\mathbf{true} & \Rightarrow & \neg x \vee F \end{array}}$$

Note that these transformation rules are derived from the quantified interpretation of the fixpoint form of $\mathcal{S}$. The fixpoint form is

$$P\,\mathcal{S}\,Q \;=\; \mu\xi.\;\; \circledcirc(Q \vee (P \wedge \xi))$$

Quantifiers can be used to simulate such fixpoints [1]. For example, the above fixpoint can be represented in QPTL as follows.

$$\underbrace{\exists x.\;\Box(x \Leftrightarrow\; \circledcirc(Q \vee (P \wedge x)))}_{\text{Existence}} \;\wedge\; \underbrace{\forall y.\;\Box(y \Leftrightarrow\; \circledcirc(Q \vee (P \wedge y))) \Rightarrow (x \Rightarrow y)}_{\text{Minimality}}$$

The 'existence' part of this formula states that there is some proposition, x, which satisfies the fixpoint formula $x \Leftrightarrow\; \circledcirc(Q \vee (P \wedge x))$ at all points in time, while the 'minimality' part ensures that this particular proposition is the least solution of the fixpoint. (Note that $x \sqsubseteq y$ if, and only if, $x \Rightarrow y$ is valid.) In this instance of METATEM, models have a finite past, and so the only difference between the maximal and minimal fixpoints is their behaviour at the beginning of time. Thus, any y found will have exactly the same valuation as x and the above definition can be simplified to

$$\exists x.\;\Box(x \Leftrightarrow\; \circledcirc(Q \vee (P \wedge x)))$$

giving the derivation of the new METATEM rules.

Because of the finite past, the rules for the weak since, the *zince* operator ($\mathcal{Z}$) are very similar to those for the (strong) since operator, the only difference being the value of the new proposition symbol, x, at the beginning of time.

$$\frac{\begin{array}{rcl} \mathcal{P}(A\,\mathcal{Z}\,B) & \Rightarrow & F \end{array}}{\begin{array}{rcl} \bullet(B \vee (A \wedge x)) & \Leftrightarrow & x \\ \mathcal{P}(x) & \Rightarrow & F \end{array}}$$

$$\frac{\begin{array}{rcl} A\,\mathcal{Z}\,B & \Rightarrow & F \end{array}}{\begin{array}{rcl} \bullet(B \vee (A \wedge x)) & \Leftrightarrow & x \\ \bullet\,\mathbf{true} & \Rightarrow & \neg x \vee F \end{array}}$$

As the ■ and ♦ operators are defined in terms of $\mathcal{S}$ and $\mathcal{Z}$, the transformation of these operators is subsumed by the above rules. The final rule in this section deals with the transformation of multiple last-time operators. Rather than give rules for the various combinations of last-time operators, we will give one rule that is parameterised by the last-time operators used:

$$\frac{\begin{array}{rcl}\mathcal{P}(\mathcal{L}A) & \Rightarrow & F\end{array}}{\begin{array}{rcl}\mathcal{L}A & \Leftrightarrow & z \\ \mathcal{P}(z) & \Rightarrow & F\end{array}}$$

4.2 Eliminating Future-Time Operators

When eliminating future-time operators, we take the $\Diamond$ operator as primitive. Thus, we use the following equivalences to remove the $\Box$ and $\mathcal{U}$ operators.

$$\begin{array}{rcl}\Box A & \equiv & A\,\mathcal{W}\,\textbf{false} \\ A\,\mathcal{U}\,B & \equiv & (A\,\mathcal{W}\,B \wedge \Diamond B)\end{array}$$

The following transformation rules can be used to remove the other future-time operators, $\mathcal{W}$ and $\bigcirc$, and to simplify $\Diamond$-formulae so that each $\Diamond$ operator is only applied to a literal, rather than a general temporal formula. First, we consider the removal of the 'unless' operator $\mathcal{W}$':

$$\frac{\begin{array}{rcl}P & \Rightarrow & \mathcal{F}(A\,\mathcal{W}\,B)\end{array}}{\begin{array}{rcl}P & \Rightarrow & \mathcal{F}(B \vee (A \wedge y)) \\ \circledcirc y & \Leftrightarrow & B \vee (A \wedge y)\end{array}}$$

The following transformation can be applied to ensure that the B in any $\Diamond B$ is a literal.

$$\frac{\begin{array}{rcl}P & \Rightarrow & \mathcal{F}(\Diamond A)\end{array}}{\begin{array}{rcl}P & \Rightarrow & \mathcal{F}(\Diamond z) \\ \bullet\,\textbf{true} & \Rightarrow & (\neg z \vee A) \wedge (z \vee \neg A)\end{array}}$$

Next-time operators can be removed by introducing new propositions and, effectively, replacing the next-time operators by last-time operators.

$$\frac{\begin{array}{rcl}P & \Rightarrow & \mathcal{F}(\bigcirc A)\end{array}}{\begin{array}{rcl}P & \Rightarrow & \mathcal{F}(w) \\ \circledcirc w & \Leftrightarrow & A\end{array}}$$

4.3 Manipulating METATEM Rules

As well as the transformation rules used for the elimination of temporal operators, simpler transformations can also be applied to set of METATEM rules. Rather than giving the full set of rules, we will only describe those that are used in the transformation of METATEM programs to a normal form (see §6.1).

First, we give simple transformation rules that 'push' negation operators to the propositions, ensuring that all negations only appear when applied to propositions. These rules are analogous to the rewrite rules used to transform formulae of classical logic into *Negation Normal Form* (NNF) used in classical logics. As these rules are so simple, we will just give them as rewrites.

$$
\begin{array}{lcl}
\{\neg\mathbf{true}\} & \longrightarrow & \{\mathbf{false}\} \\
\{\neg\mathbf{false}\} & \longrightarrow & \{\mathbf{true}\} \\
\{\neg(A \wedge B)\} & \longrightarrow & \{\neg A \vee \neg B\} \\
\{\neg(A \vee B)\} & \longrightarrow & \{\neg A \wedge \neg B\} \\
\{\neg\neg A\} & \longrightarrow & \{A\} \\
\{\neg(A\,\mathcal{U}\,B)\} & \longrightarrow & \{(\neg B)\,\mathcal{W}\,(\neg B \wedge \neg A)\} \\
\{\neg(A\,\mathcal{W}\,B)\} & \longrightarrow & \{(\neg B)\,\mathcal{U}\,(\neg B \wedge \neg A)\} \\
\{\neg\Diamond A\} & \longrightarrow & \{(\neg A)\,\mathcal{W}\,\mathbf{false}\} \\
\{\neg\bigcirc A\} & \longrightarrow & \{\bigcirc\neg A\}^{3} \\
\{\neg(A\,\mathcal{S}\,B)\} & \longrightarrow & \{(\neg B)\,\mathcal{Z}\,(\neg B \wedge \neg A)\} \\
\{\neg(A\,\mathcal{Z}\,B)\} & \longrightarrow & \{(\neg B)\,\mathcal{S}\,(\neg B \wedge \neg A)\} \\
\{\neg\bullet A\} & \longrightarrow & \{\circledcirc\neg A\} \\
\{\neg\circledcirc A\} & \longrightarrow & \{\bullet\neg A\}
\end{array}
$$

We now present rules that are analogous to the transformations carried out in the production of Disjunctive and Conjunctive Normal Forms for classical logics. Note that the following rules can be used in either direction, but are given in the form that they are used in the production of a normal form (§6.1).

$$\frac{\mathcal{L}(A \wedge (B \vee C)) \;\Rightarrow\; F}{\mathcal{L}((A \wedge B) \vee (A \wedge C)) \;\Rightarrow\; F}$$

$$\frac{\mathcal{L}(A) \wedge \mathcal{L}(B) \;\Rightarrow\; F}{\mathcal{L}(A \wedge B) \;\Rightarrow\; F}$$

$$\frac{\mathcal{L}(A) \vee \mathcal{L}(B) \;\Rightarrow\; F}{\mathcal{L}(A \vee B) \;\Rightarrow\; F}$$

$$\frac{\begin{array}{rcl}\circledcirc A & \Rightarrow & F \\ \circledcirc B & \Rightarrow & F\end{array}}{\circledcirc(A \vee B) \;\Rightarrow\; F}$$

The second and third transformations are applicable only if the last-time formulae being combined have the same last-time operators. In the case where different last-time operators are combined, the following rules are used.

$$\frac{\circledcirc A \wedge \bullet B \;\Rightarrow\; F}{\circledcirc(A \wedge B) \;\Rightarrow\; F}$$

$$\frac{\circledcirc A \vee \bullet B \;\Rightarrow\; F}{\bullet(A \vee B) \;\Rightarrow\; F}$$

[3] Note that as all our temporal models have infinite futures, this is a valid rewrite rule.

Though the formal justification for these rules will be given later, we will give an informal description of the first one. At the beginning of time $\boxdot A \wedge \bullet B$ will be false (as $\boxdot A$ is false there for any A) and at any later time point, $\bullet B$ will act just as $\boxdot B$ would. Thus, we can replace $\bullet B$ by $\boxdot B$ and merge this with $\boxdot A$ to give $\boxdot(A \wedge B)$.

Any, weak last-time operators ($\bullet$) that are not applied to **false** are also removed from the past-time component:

$$\frac{\begin{array}{rcl} \bullet A & \Rightarrow & F \end{array}}{\begin{array}{rcl} \boxdot A & \Rightarrow & F \\ \bullet \textbf{false} & \Rightarrow & F \end{array}}$$

The conversion rules for the future-time components of each clause are considered next. As the only operators remaining in this part are $\wedge$, $\vee$, $\diamondsuit$, and $\neg$, and as both $\diamondsuit$ and $\neg$ operators only apply to literals, only combinations of conjuncts and disjuncts need to be rewritten. The transformation rules used for this are as follows.

$$\frac{\begin{array}{rcl} P & \Rightarrow & A \vee (B \wedge C) \end{array}}{\begin{array}{rcl} P & \Rightarrow & (A \vee B) \wedge (A \vee C) \end{array}}$$

$$\frac{\begin{array}{rcl} P & \Rightarrow & A \wedge B \end{array}}{\begin{array}{rcl} P & \Rightarrow & A \\ P & \Rightarrow & B \end{array}}$$

To ensure that at most one $\diamondsuit$ operator appears in the future-time component of each clause, the following transformation rule is required.

$$\frac{\begin{array}{rcl} P & \Rightarrow & \diamondsuit A \vee \diamondsuit B \end{array}}{\begin{array}{rcl} P & \Rightarrow & \diamondsuit(A \vee B) \end{array}}$$

Note that, when this rule is applied, the transformation given on page 13 will also need to be applied to ensure that '$\diamondsuit$' is only applied to a literal.

4.4 Resolution Transformations

The transformations characterised as *Resolution Transformations* can either be semantics preserving, or satisfiability preserving, depending on whether the new rules generated are added to the set of rules (semantics preserving) or replace certain rules (satisfiability preserving). The basic resolution rule is satisfiability preserving:

$$\frac{\begin{array}{rcl} P & \Rightarrow & A \vee \neg b \\ Q & \Rightarrow & C \vee b \end{array}}{\begin{array}{rcl} P \wedge Q & \Rightarrow & A \vee C \end{array}}$$

Here, b is a proposition. As with classical resolution, various strategies for the application of this rule can be employed, and simplification rules can be used during, and after, its application.

Simplification The basic simplification rules are

$$\frac{\circledcirc\,\textbf{false} \;\Rightarrow\; F}{}$$

$$\frac{P \;\Rightarrow\; \textbf{true}}{}$$

In both these cases, the rules matching the antecedent are deleted. The removal of such rules is obvious, since $\circledcirc\,\textbf{false}$ can never be satisfied, and $P \Rightarrow \textbf{true}$ is always satisfiable.

Similarly, subsumption rules may be applied. Along with the standard (classical) subsumption, the following rule for subsumption between rules can be applied.

$$\frac{\begin{array}{l} P \;\Rightarrow\; F \\ Q \;\Rightarrow\; G \end{array}}{Q \;\Rightarrow\; G}$$

This rule can only be applied if the conditions, $\vdash P \Rightarrow Q$ and $\vdash G \Rightarrow F$ are derivable.

If, after simplification, a formula of the form

$$\circledcirc\, R \;\Rightarrow\; \textbf{false}$$

has been derived, then the following transformation rule can be invoked.

$$\frac{\circledcirc\, R \;\Rightarrow\; \textbf{false}}{\bullet\,\textbf{true} \;\Rightarrow\; \neg R}$$

5 Correctness of the Transformations

In this section we will give a selection of proofs of consistency for the transformations described in this paper. Rather than give a proof for every transformation, we will outline the proof strategy for the two main types of transformation and give proofs for only some of the difficult cases.

We will first consider semantics preserving transformations, and the properties that need to be established for them.

5.1 Correctness of Semantics Preserving Transformations

For a semantics preserving transformation τ, we must show that the models of an arbitrary formula, A, are the same as those of $\tau(A)$, apart from the valuation given to any new proposition symbol. Semantically, this condition can be written as follows. (Note that we treat both A and $\tau(A)$ as temporal formulae, i.e. as formulae representing sets of METATEM rules.)

> For all A and for all σ,
> $(\sigma, 0) \models A$ iff there exists a σ' such that $\sigma \sim^x \sigma'$ and $(\sigma', 0) \models \tau(A)$.

Here, 'x' is the new proposition introduced and $\sigma \sim^x \sigma'$ if, and only if, σ differs from σ' in at most the valuation given to x.

This criterion can be characterised logically in our temporal logic extended with quantification over propositions (QPTL) as $\vdash_{\text{QPTL}} A \Leftrightarrow \exists x.\ \tau(A)$.

In proving the consistency of semantics preserving transformations we use the fixpoint definitions of the various temporal operators and utilise the correspondence between QPTL and temporal logic with fixpoints (νtl) [20, 1]. For example, recall that we defined the $\tau_\Box$ transformation as

$$\tau_\Box(\{P \Rightarrow \Box B\} \cup \Psi) \;=\; \{P \Rightarrow (x \wedge B),\ \text{◎} x \Rightarrow (x \wedge B)\} \cup \Psi$$

To simplify the proof of semantics preservation for this transformation, we will restrict this rule to the case where Ψ is empty. Note that, as the new proposition to be introduced does not appear in Ψ, the proof given here for empty Ψ can be extended to one with non-empty Ψ.

In order to prove that the above transformation preserves semantics in the case where Ψ is empty we must establish that, for any formula A, $\vdash_{\text{QPTL}} A \Leftrightarrow \exists x.\ \tau_\Box(A)$. Below we outline the structure of this proof.

Theorem 1 *For an arbitrary* METATEM *rule, A,* $\vdash_{\text{QPTL}} A \;\Leftrightarrow\; \exists x.\ \tau_\Box(A)$.

Proof For a given formula, A, we prove each direction separately, i.e.,

1. $\vdash_{\text{QPTL}} A \;\Rightarrow\; \exists x.\ \tau_\Box(A)$

   To prove this, we simply replace $\Box B$ by its expansion in terms of existential quantification.

$$\Box(P \Rightarrow \Box B) \;\Rightarrow\; \exists x.\ \Box(x \Leftrightarrow \bigcirc(x \wedge B)) \wedge \Box(P \Rightarrow (x \wedge B))$$

By applying simple rewrites to move $\Box$ through this formula, the following is produced.

$$\Box(P \Rightarrow \Box B) \;\Rightarrow\; \exists x.\ \Box((\circledcirc x \Leftrightarrow (x \wedge B)) \wedge (P \Rightarrow (x \wedge B)))$$

2. $\vdash_{\text{QPTL}} \tau_{\Box}(A) \Rightarrow A$

   To prove this, we start with the transformed rule (i.e., $\Box(\circledcirc x \Leftrightarrow (x \wedge B))$) and proceed through the following steps

   (a) $\Box(\circledcirc x \Leftrightarrow (x \wedge B)) \;\Leftrightarrow\; \Box(\circledcirc x \Leftrightarrow \Box(x \wedge B))$

   (b) $\Box(\circledcirc x \Leftrightarrow \Box(x \wedge B)) \;\Rightarrow\; \Box(\circledcirc x \Leftrightarrow \Box B)$

   (c) $\Box(\circledcirc x \Leftrightarrow \Box(x \wedge B)) \;\Rightarrow\; \Box(x \Leftrightarrow \bigcirc \Box B)$

   (d) $\Box(P \Rightarrow (x \wedge B)) \wedge \Box(x \Leftrightarrow \bigcirc \Box B) \;\Rightarrow\; \Box(P \Rightarrow \Box B)$

5.2 Correctness of Satisfiability Preserving Transformations

For a satisfiability preserving transformation, τ, we must show that if $\tau(A)$ is unsatisfiable, then A must have been unsatisfiable. There is no simple syntactic characterisation of this criterion, however, it can be described semantically as follows.

$$\begin{array}{c}\text{For all formulae A, if there exists a } \sigma \text{ such that } (\sigma, 0) \models A \\ \text{then there exists a } \sigma' \text{ such that } (\sigma', 0) \models \tau(A)\end{array}$$

Though no examples will be given here, proving the consistency of satisfiability preserving transformations is usually easier than proving similar properties of semantics preserving transformations. One reason for this is that the transformations involving the simulation of fixpoints using quantifiers are nearly all categorised as semantics preserving. Thus, the consistency of satisfiability preserving transformations can usually be established without expansion of fixpoint operators. For example, the proof that τ_R, the resolution transformation, preserves satisfiability, follows the soundness result for classical resolution.

5.3 Correctness of Determinising Transformations

For a determinising transformation, τ, we must show that the models of $\tau(A)$ are a subset of those of A for any formula A representing a set of METATEM rules. The proof of this type of property is usually the easiest of the three types of transformation described.

To prove the consistency of a transformation such as $\tau_\vee$, where

$$\tau_\vee(\{P \Rightarrow A \vee B\} \cup \Psi) \;=\; \{P \Rightarrow A\} \cup \Psi$$

we need to show that

$$\begin{array}{ll}\text{For all } \sigma, & \text{if } [(\sigma, 0) \models \Box(P \Rightarrow A) \wedge \Box \bigwedge\limits_{\xi \in \Psi} \xi] \\ & \text{then } [(\sigma, 0) \models \Box(P \Rightarrow A \vee B) \wedge \Box \bigwedge\limits_{\xi \in \Psi} \xi].\end{array}$$

This is trivially satisfied since $\vdash A \Rightarrow (A \vee B)$.

Similarly, $\tau_{\Diamond}$ can easily be shown to satisfy the correctness criterion.

Note, however, that once a determinising transformation has been applied, satisfiability is no longer guaranteed to be preserved. Thus, in the synthesis procedure outlined in §6.4, we must either be careful to choose a deterministic formula that will still be satisfied when the original formulae is satisfied, or must use a combined determinisation and satisfiability preserving transformation.

6 Applications of METATEM Program Transformation

6.1 Transforming Programs into a Normal Form

In this section, we describe one application of the transformation rules described above, namely the transformation of METATEM programs into a normal form. This normal form, called *Separated Normal Form* (SNF) is defined as follows [10].

A temporal formula in SNF is of the form

$$\Box \bigwedge_{i=1}^{n} (P_i \Rightarrow F_i).$$

Here, each P_i is a *strict* past-time temporal formula and each F_i is a (non-strict) future-time formula. Each of the '$P_i \Rightarrow F_i$' (called *rules*) is further restricted to be one of the following

$$\bullet\,\mathbf{false} \Rightarrow \bigvee_{k=1}^{r} l_k \qquad \text{(an } \textit{initial} \ \Box\text{-rule)}$$

$$\circledcirc \bigwedge_{j=1}^{m} l_j \Rightarrow \bigvee_{k=1}^{r} l_k \qquad \text{(a } \textit{global} \ \Box\text{-rule)}$$

$$\bullet\,\mathbf{false} \Rightarrow \Diamond l \qquad \text{(an } \textit{initial} \ \Diamond\text{-rule)}$$

$$\circledcirc \bigwedge_{j=1}^{m} l_j \Rightarrow \Diamond l \qquad \text{(a } \textit{global} \ \Diamond\text{-rule)}$$

where each l_j, l_k, or l is a literal.

Sets of rules in SNF incorporate the following restriction. For each literal, l, there are at most two rules with a single occurrence of l (i.e., not as part of a disjunction) as their future-time component. If there are two such rules, one must be initial, while the other must be global. This ensures that there is, at most, one rule defining the initial value of l and, at most, one rule constraining l after the beginning of time.

The production of SNF from an arbitrary set of temporal formulae (each of which have been separated into past, present and future), can be achieved by applying the appropriate transformation rules in the following sequence.

1. 'Push' negation through the formulae until negations only apply to propositions.
2. Remove $\blacksquare$, $\blacklozenge$, $\mathcal{S}$, $\mathcal{Z}$, $\bigcirc$, $\Box$, $\mathcal{W}$, and $\mathcal{U}$.
3. Remove multiple last-time operators and ensure that each $\Diamond$ operator only applies to a literal.
4. Rewrite the remaining formulae into SNF (effectively a form of CNF). This is achieved using the rules described in §4.3.

Note that, as the models for our logic have infinite futures, temporal operators that are defined as least fixpoints, such as '$\Diamond$' and '$\mathcal{U}$', can not be completely eliminated. However, as any such operator can often be represented by any other operator defined as a least fixpoint, the number of operators defined in this way can be reduced to one. For example, in SNF, any formula can be transformed so that it contains only one temporal operator defined as a least fixpoint, namely the '$\Diamond$' operator.

6.2 Compilation of METATEM Programs

One immediate application of the transformation into SNF described above is as a preprocessing stage in a compilation process. SNF provides a form of METATEM rule that can be easily translated into an I/O automaton [14]. Such preprocessing is used in the compilation process described in [8].

6.3 Temporal Resolution

The SNF rule form is also used in the temporal resolution process described in [10]. The resolution rules described above are used for *non-temporal* resolution, while a specialised transformation rule implementing a form of temporal resolution is applied to the $\Diamond$-rules. Though the temporal resolution rule is a complex transformation with several side conditions, the application of this rule can be implemented by representing the set of formulae as a graph and searching for appropriate paths through the graph.

The temporal resolution process terminates and, if one of the following rules have been generated, then the original set of rules was unsatisfiable.

$$\begin{array}{lrcl} a) & \bullet\,\textbf{false} & \Rightarrow & \textbf{false} \\ b) & \bullet\,\textbf{true} & \Rightarrow & \textbf{false} \\ c) & \circledcirc\,\textbf{true} & \Rightarrow & \textbf{false} \end{array}$$

Otherwise the temporal formula is satisfiable. For a more detailed description of this approach, see [10].

6.4 Deriving Deterministic METATEM Programs

One of the major applications, within the METATEM framework, of the type of syntactic transformations described above is in the synthesis of deterministic programs, i.e. programs which do not require backtracking to previous states, from general (non-deterministic) ones. A method to perform automatic synthesis within METATEM is discussed in section 6.4.1. It is intended, however, that METATEM will also allow some form of controlled synthesis. A brief argument for this approach is given in section 6.4.2

6.4.1 Automatic synthesis

One approach to the synthesis problem consists in expressing the specification in the form of a model-accepting automata, and then transforming it into a deterministic transducer, i.e. an

automata which, at a given state, produces a unique valuation of the component propositions for each possible behaviour of the environment up to and including that state. This approach has been described by Pnueli and Rosner [17, 18]. Our approach to the synthesis problem is to transform a general temporal specification (that has been written in the form of METATEM rules) into a deterministic, fair set of rules that can be treated as an implementation of the specification. The transformation, described in [15] for the case where the specification is expressed in propositional temporal logic, may easily be adapted to specifications expressed in METATEM[4]. We will here give an outline of the approach and comment on its applicability. The steps of the transformation process are:

1. The transformation of the program into SNF.
   (see §6.1.)

2. The resolution of any conflicts within the rules.
   (See §4.4.)

3. The 'fair' scheduling of the eventualities.

The scheduling of the eventualities described in [15] and briefly presented below involves the addition of an explicit scheduling to the specification. All the eventuality symbols are removed. A more efficient way, in which part of the scheduling is performed at run-time is suggested later in this section.

Adding the scheduling to the specification. The general idea behind the determinisation algorithm consists in enforcing the satisfaction of only one outstanding eventuality at a time and ensuring that all the eventualities are eventually satisfied. Typically, this is implemented by attempting to satisfy the oldest (or one of the oldest if there are many of the same age) outstanding eventuality first.

Generally, given an eventuality formula

$$\bigcirc p \Rightarrow \Diamond l_0, \tag{1}$$

the formulae which constrain the satisfaction of the eventuality literal l_0 are the formulae which may be written in the form:

$$\bigcirc \neg l_{i,j} \Rightarrow \neg l_{i-1} \vee e_{i,j} \qquad (0 < i \leq n)$$

where l_k stands for $\bigwedge_j l_{k,j}$, and $e_{i,j}$ is a state formula involving only environment literals. We will restrict the following discussion to the case where no environment literal appearing in a constraint (or more precisely in the component $e_{i,j}$ of a constraint) also appears in its negated form in another constraint[5]. If the specification is implementable (i.e. if it does not constrain the future behaviour of the environment) the constraints must be considered for the worst

[4]Note however that the operators $\mathcal{S}$ and $\mathcal{Z}$ have a slightly different meaning in the logic used in [15]

[5]Work is still in progress towards producing a 'reasonable' algorithm for the general case

possible behaviour of the environment. In the particular case concerned here, they may thus be expressed as:

$$\boxdot \neg l_i \Rightarrow \neg l_{i-1} \qquad (0 < i \leq n).$$

The sequence of constraints is such that either no formula constraining l_n may be found, or l_{n+1} implies l_m for some $m \leq n$.

The suggested transformation consists in replacing the formulae such as (1) by the conjunction of the following formulae, rewritten into SNF:

$$(\mathit{outs}(l_0) \wedge \mathit{priority}(l_0) \wedge \boxdot(\bigvee_{1 \leq j < i} \neg l_j) \wedge l_i) \Rightarrow l_{i-1} \qquad (0 < i \leq n)$$

where $\mathit{outs}(l_0)$ stands for $\neg l_0 \, \mathcal{S} \, p$ and $\mathit{priority}(l_0)$ is a formula specifying that l_0 is the outstanding literal to be satisfied next. It is shown in [15] that, if the specification is implementable, the formula

$$\mathit{outs}(l_0) \wedge \mathit{priority}(l_0) \Rightarrow \bigvee_{i \leq n} l_i$$

may be added to the specification without introducing inconsistencies.

Run-time scheduling. The full inclusion of the scheduling information within the determinised program, as described above, is not completely satisfactory. First, only one outstanding eventuality is scheduled at any one time while, generally, the conditions for satisfying more than one eventuality may occur together. Next, the formula specifying when a literal is to be scheduled may be quite complex. In the case where the oldest outstanding literal has to be satisfied first, for instance, $\mathit{priority}(l_0)$ involves a pairwise comparison of the lengths of time during which l_0 and each of the other outstanding literals have been unsatisfied.

An alternative solution is to defer part of the scheduling to run time, by making use of the (minimal) scheduling facilities already present in METATEM. This again involves adding a scheduling formula to the original program, but

a) this formula will not, in general, be as complex as the formula specifying the complete scheduling of all the eventualities, and,

b) the formula will reduce the problem of scheduling all the eventualities to a form that the basic METATEM execution mechanism is able to cope with.

In the case where the environment components of the constraints may be removed, the formula

$$\mathit{outs}(l_0) \Rightarrow \diamondsuit l$$

may be used to replace (1) after the specification, together with the relevant part of the definition of l,

$$\bullet \mathit{true} \Rightarrow \neg l \vee \bigvee_{i \leq n} (\bigwedge_{j \leq i} \bigcirc^{i-j} l_j),$$

has been transformed into SNF. In this way, as soon as METATEM 'decides' to satisfy l_0, all the conditions necessary for its eventual satisfaction are enforced. Whenever possible, METATEM will also enforce the conditions for the satisfaction of other outstanding literals.

Further work in this area is under way.

6.4.2 Controlled synthesis

Using METATEM, we can adopt a transformational approach to synthesis. One of the problems with the automatic synthesis approach is that there is little control of the final implementation from the specifier. Using METATEM we can, by invoking meta-level procedures, guide the transformation of the specification into a suitable implementation. At each transformation step we have the option of testing the current system by executing it and interactively manipulating the program.

7 Conclusions and Future Work

In this paper, we have described a range of transformation rules for METATEM programs. Many of these transformations are simple and can be seen as logical transformations between equivalent temporal formulae. We have outlined several applications of these transformations in propositional METATEM, particularly in the synthesis of deterministic METATEM programs from arbitrary temporal specifications. A version of the determinisation procedure outlined briefly in §6.4 and in more detail in [15] has been implemented in ML and the initial results are encouraging.

As mentioned above, the synthesis procedure is just the first attempt at this problem. Further research will no doubt uncover more efficient or simple methods.

The main line of research to be pursued in relation to transformation in general is in the development of transformation rules for *first-order* METATEM programs. Initial work in this area has developed a variety of SNF for first-order temporal formulae and a small set of transformations which can be used to transform a first-order formula in METATEM rule form into first-order SNF [9]. Work is also proceeding into the extension of the syntactic determinisation procedure described in [15] and §6.4 to a subset of first-order temporal logic [16].

8 Acknowledgements

The work described in this report was supported jointly by ESPRIT under Basic Research Action 3096 (SPEC), and by SERC under Research Grant GR/F 30123. Both authors would like to thank Brian Monahan for his helpful comments on an earlier draft of this report.

References

[1] B. Banieqbal and H. Barringer.
Temporal Logic with Fixed Points.
In *Proceedings of the Colloquium on Temporal Logic and Specification (LNCS Vol. 398)*, pages 62–74, Altrincham, U.K., 1989. Springer-Verlag.

[2] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens.
METATEM: A Framework for Programming in Temporal Logic.
In *REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Mook, Netherlands, June 1989.
(Published in *Lecture Notes in Computer Science*, volume 430, Springer Verlag).

[3] H. Barringer, M. Fisher, D. Gabbay, and A. Hunter.
Meta-Reasoning in Executable Temporal Logic.
In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, Cambridge, Massachusetts, April 1991. Morgan Kaufmann.

[4] R. M. Burstall and J. Darlington.
A Transformation System for Developing Recursive Programs.
ACM Journal, 24(1):44–67, 1977.

[5] John Darlington.
Program Transformation and Synthesis: Present Capabilities.
Artificial Intelligence, 16, 1981.

[6] John Darlington.
Program Transformation.
BYTE, 10(8):201–215, August 1985.

[7] M. Fisher and H. Barringer.
Concurrent METATEM Processes — A Language for Distributed AI.
In *Proceedings of the European Simulation Multiconference*, Copenhagen, Denmark, June 1991.

[8] M. Fisher and T. Longshaw.
Compiling METATEM Programs using I/O-Automata.
(In preparation.), 1992.

[9] Michael Fisher.
A Normal Form for First-Order Temporal Formulae.
In *Proceedings of Eleventh International Conference on Automated Deduction (CADE)*. Springer-Verlag, 1991.

[10] Michael Fisher.
A Resolution Method for Temporal Logic.
In J. Mylopoulos and R. Reiter, editors, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Sydney, Australia, August 1991. Morgan Kaufman.

[11] Dov 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 (LNCS Volume 398)*, pages 402–450, Altrincham, U.K., 1989. Springer-Verlag.

[12] Susan Haack.
Philosophy of Logics.
Cambridge University Press, 1978.

[13] Cliff B. Jones.
Systematic Software Development Using VDM.
Prentice Hall International, Englewood Cliffs, NJ, 1986.

[14] N. A. Lynch and M. R. Tuttle.
An Introduction to Input/Output Automata.
CWI Quarterly, 2(3):219–246, September 1989.
Centre for Mathematics and Computer Science, Amsterdam.

[15] Philippe A. Noël.
A Method for the Determinisation of Propositional Temporal Formulae.
METATEM Project Report, Department of Computer Science, University of Manchester, 1991.
(To appear in Proceedings of LOPSTR'91).

[16] Philippe A. Noël.
Determinisation of First-order Monadic Temporal Formulae.
(In preparation), January 1992.

[17] 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.

[18] 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.

[19] H. Tamaki and T. Sato.
A Transformation System for Logic Programs which Preserves Equivalence.
Technical Report TR-018, ICOT, Minato-ku, Tokyo, Japan, August 1983.

[20] Pierre Wolper.
Synthesis of Communicating Processes from Temporal Logic Specifications.
PhD thesis, Stanford University, 1982.