Presented at the IFIP Working Conf. on Programming Concepts, Methods and Calculi (PROCOMET '94), 6-10 June 1994, San Miniato - Italy.
Included in *Programming Concepts, Methods and Calculi*, E.-R. Olderog (ed.), IFIP Transactions, Vol. A-56, North-Holland, 1994, 307–326.

Some Very Compositional Temporal Properties

Ben Moszkowski[a]

[a]Department of Computing Science, University of Newcastle upon Tyne,
Newcastle NE1 7RU, Great Britain
e-mail: Ben.Moszkowski@ncl.ac.uk

A logic for reasoning about sequential and parallel behavior must support some form of compositionality. That is, much of the proof of a system should be decomposible into proofs of its parts. We discuss some important and easily described classes of properties which are readily imported and exported between temporal scopes. We call such properties *very compositional* since they support a methodology of specification and proof which is especially modular and reusable. Our presentation uses Interval Temporal Logic as the notation in which all behavior is described and proved. We give a powerful and elegant ITL proof system. It has been used by us to rigorously prove hundreds of theorems and derived inference rules, including a number for parallel systems involving message-passing, timing constraints and shared write-access. We believe that very compositional properties will be of interest to anyone involved with the specification and verification of computer systems.

Keyword Codes: F.4.1; I.2.3
Keywords: Mathematical Logic; Deduction and Theorem Proving

1. INTRODUCTION

The notion of compositionality is very important in computer science. It facilitates the modular design and maintenance of complex systems and the reuse of the intellectual and physical resources invested in them. Successful examples of its use include modular hardware design, subroutine libraries, object-oriented programming and open systems. Compositionality is also of great value in programming logics. A major reason for the success of Hoare's axiomatic system [1] for sequential programs is the ease with which the proofs of a program's pieces can be combined together. We have been using Interval Temporal Logic [2–5] to investigate various forms of parallel behavior. In the course of our research, we have identified certain classes of properties that are especially suited for compositional temporal specifications and proofs.

The basic nature of these compositional properties is not hard to grasp. In order for the reader to better appreciate their significance, we now give a brief, relatively informal presentation of them within the framework of ITL. Basically ITL is a logic for reasoning about discrete intervals or periods of time and consists of conventional logic constructs as well the three operators *skip*, ; (*chop*) and * (*chop-star*). An interval is a finite, nonempty

sequence of states. The length of an interval is defined to be the number of states in the interval minus one. A precise definition of ITL's syntax, semantics and proof system is given later on.

Suppose we have two formulas $T;T'$ and T^*. The first one $T;T'$ consists of the sequential composition of subformulas T and T'. The formula $T;T'$ is true on an interval iff the interval can be split into two subintervals sharing a common state and in which subformulas T and T' are respectively true. The formula T^* denotes the sequential iteration of the specification T some finite number of times and is similar to the Kleene star of regular expressions. We will not give any more details about T and T' at this time. Consider the following proof rule:

$$\frac{\begin{array}{ll}\vdash & w \wedge T \supset \mathit{fin}\ w', \\ \vdash & w' \wedge T' \supset \mathit{fin}\ w''\end{array}}{\vdash \quad w \wedge (T;T') \supset \mathit{fin}\ w''.}$$

Here w, w' and w'' are formulas in conventional first-order logic containing no temporal operators and describing properties of individual states. The turnstyle $\vdash$ means that the formula to its right is provable in our axiom system. Note that a conjunction of formulas is true on an interval iff each formula is true. The temporal operator *fin* is definable in ITL and lets us examine an interval's final state. The first lemma states that if w is true in an interval's initial state and T is true on the interval then w' is true in the final state. The rule shows how to compose two such lemmas proved about input-output behavior of T and T' into a corresponding lemma for $T;T'$. Here is an analogous proof rule for T^*:

$$\frac{\vdash \quad w \wedge T \supset \mathit{fin}\ w}{\vdash \quad w \wedge T^* \supset \mathit{fin}\ w.}$$

The rough meaning of the rules so far given should be reasonably clear. They and all the other proof rules shown here can be formally derived in our proof system presented later in section 3. Other similar rules can be proved for *if* and *while* constructs.

So far we have dealt solely with input-output properties. The following are generalizations of the previous rules to extract more information about behavior in intermediate states:

$$\frac{\begin{array}{ll}\vdash & w \wedge T \supset U \wedge \mathit{fin}\ w', \\ \vdash & w' \wedge T' \supset U' \wedge \mathit{fin}\ w''\end{array}}{\vdash \quad w \wedge (T;T') \supset (U;U') \wedge \mathit{fin}\ w''} \qquad \frac{\vdash \quad w \wedge T \supset U \wedge \mathit{fin}\ w}{\vdash \quad w \wedge T^* \supset U^* \wedge \mathit{fin}\ w.} \qquad (1), (2)$$

Using these, we can compositionally transform one specification into another. This is because we reduce the overall proof into proofs for the subformulas.

Now consider the following two proof rules which are in general *not* sound:

$$\frac{\begin{array}{ll}\vdash & w \wedge S \wedge T \supset U \wedge \mathit{fin}\ w', \\ \vdash & w' \wedge S \wedge T' \supset U \wedge \mathit{fin}\ w''\end{array}}{\vdash \quad w \wedge S \wedge (T;T') \supset U \wedge \mathit{fin}\ w''} \qquad \frac{\vdash \quad w \wedge S \wedge T \supset U \wedge \mathit{fin}\ w}{\vdash \quad w \wedge S \wedge T^* \supset U \wedge \mathit{fin}\ w.} \qquad (3), (4)$$

The formulas S and U correspond to some general temporal "ongoing" properties. For instance, the first lemma in the rule on the left says that if w is initially true and S is true

when T is performed, it follows that U is true and w' is true in the final state. Unlike rules 1 and 2 above, these rules do not work for arbitrary formulas S and U. The two rules below are in fact the best we can do without imposing any restrictions:

$$\frac{\vdash\ w \wedge S \wedge T \supset U \wedge \mathit{fin}\ w',\quad \vdash\ w' \wedge S \wedge T' \supset U \wedge \mathit{fin}\ w''}{\vdash\ w \wedge ((S \wedge T);(S \wedge T')) \supset (U;U) \wedge \mathit{fin}\ w''} \qquad \frac{\vdash\ w \wedge S \wedge T \supset U \wedge \mathit{fin}\ w}{\vdash\ w \wedge (S \wedge T)^* \supset U^* \wedge \mathit{fin}\ w.} \quad (5),(6)$$

These are in fact instances of rules 1 and 2 given earlier. Thus rules 3 and 4 provide a degree of compositionality much better than rules 5 and 6 and we therefore call them *very compositional.*

The question remains regarding what the minimal requirements are for S and U for the rules to work. Intuitively, it can be seen that we need the following theorems to be provable for showing that S can be imported into subintervals:

$$\vdash \quad S \wedge (T;T') \quad \supset \quad (S \wedge T);(S \wedge T'), \qquad \vdash \quad S \wedge T^* \quad \supset \quad (S \wedge T)^*.$$

Here are analogous theorems for exporting sequences of the formula U:

$$\vdash \quad U;U \quad \supset \quad U, \qquad \vdash \quad U^* \quad \supset \quad U.$$

We say that S is *temporally importable* and similarly that U is *temporally exportable.*

One way to formally and concisely describe temporally importable and exportable properties is through fixpoints. Let the formula $\boxdot S$ be defined to be true for an interval of time exactly if S is true in all subintervals. A simple and formal definition of $\boxdot$ will be given later. We simply need to prove that S and U are solutions to the following equivalences:

$$\vdash \quad S \equiv \boxdot S, \qquad \vdash \quad U \equiv U^*.$$

Like the proof rules which use them, such properties are called *very compositional* since lemmas developed with them are especially combinable. The property "*The interval has length less than* 100" is an example of a fixpoint of $\boxdot$ since the property is true for an interval if and only if it is true for all subintervals. Another example is "*The variable K always equals* 1." However, neither of these properties is a fixpoint of *chop-star*. The property "*The interval has even length*" is a fixpoint of *chop-star* since whenever there is a sequence of adjacent subintervals each satisfying the property, the overall interval also has it since the sum of some even numbers is itself even. The property "*The variable I's initial and final values in the interval are equal*" is also a fixpoint of *chop-star*. Neither example is a fixpoint of $\boxdot$. Incidentally, every formula expressible as $\boxdot S$ for some arbitrary S is a fixpoint of $\boxdot$. Likewise, any formula of the form S^* for some arbitrary S is a fixpoint of *. The following theorems describe this:

$$\vdash \quad \boxdot S \equiv \boxdot \boxdot S, \qquad \vdash \quad S^* \equiv S^{**}.$$

Let us now turn to properties that are *both* temporally importable and exportable. Such properties are especially convenient since they can be exported from one proof and then readily imported in later proofs. Examples of this are the properties "*The variable A's*

value remains stable throughout the interval" and "*The variable K keeps increasing by* 1 *from each state to its immediate neighbor*." These properties are fixpoints of both $\boxed{a}$ and *chop-star*. Let the formula *keep* S be true if S is true in all subintervals of length 1 (i. e., having exactly two states). We can express *keep* S using $\boxed{a}$ and *skip*:

$$keep\ S \quad \stackrel{\text{def}}{\equiv} \quad \boxed{a}(skip\ \supset\ S).$$

This works because *skip* is defined to be true exactly for intervals having length 1 (i. e., two states). It is easy to prove that a fixpoint of *keep* is also a fixpoint of $\boxed{a}$ and *. It therefore follows that fixpoints of *keep* are both importable and exportable. In fact, we can show the converse that any property that is a fixpoint of both $\boxed{a}$ and * is also a fixpoint of *keep* . Thus, in a important sense *keep* characterizes a major class of properties that are both importable and exportable. This can be formalized in the following way:

$$\vdash \quad S \equiv \boxed{a}\, S, \quad \vdash \quad S \equiv S^* \quad \text{iff} \quad \vdash \quad S \equiv keep\ S.$$

The proof of this makes use of the following interesting lemma for expressing *keep* using *chop-star*:

$$\vdash \quad keep\ S \equiv (skip \wedge S)^*.$$

Further note that if S and T are both fixpoints of $\boxed{a}$, then so is their conjunction $S \wedge T$. This also applies to the conjunction of two fixpoints of *keep* . Also, any formula of the form *keep* S for some S is a fixpoint of *keep* .

Readers may be interested in comparing our techniques for sequential composition with Stirling's proof system [6] for an extended Hoare logic. It is based on the rely-guarantee approach of Jones [7] and is intended for reasoning about state invariants that hold true throughout a concurrent computation. Francez and Pnueli [8] introduce *interface predicates* for handling parallel composition. See Cousot [9] for a discussion about compositional proof systems for concurrency.

2. OVERVIEW OF BASIC INTERVAL TEMPORAL LOGIC

Let us examine the syntax and semantics of ITL. If the reader prefers, he can initially skip this section and the next one about a practical ITL proof system and proceed to section 4 on applications.

2.1. Syntax

2.1.1. Alphabet and sorts

The first thing to consider is the alphabet of symbols used in building constructs. We assume the following distinct sets:

- Static variables: a, b, c, ...
- State variables: A, B, C, ...
- Function symbols: f, g, ...
- Predicate symbols: p, q, ...
- Equality: $=$.
- Conventional logical symbols: $\neg$, $\wedge$, $\forall$, :, (and)
- Definite description symbol: $\imath$
- Temporal logical symbols: *skip* and ; and *

Each function and predicate symbol has an associated arity. In practice we use constants (nullary functions) such as 0 and 1, as well as functions such as $+$ and $\div$ and predicates such as $=$ and $<$.

We assume sorts referenced by the positive integers 1, 2, ... Each variable v is associated with one sort $\hat{v}$. Each n-ary predicate symbol p has n associated sorts $\hat{p}_1$, ..., $\hat{p}_n$ for its parameters. Similarly each n-ary function symbol has $n+1$ sorts $\hat{f}_1$, ..., $\hat{f}_{n+1}$. The first n sorts are for the parameters and the $n+1$-st sort is for the function's range.

2.1.2. Syntax of expressions

Expressions are built inductively as follows:

- Static variables (lower case): a, b, c, ...
- State variables (upper case) : A, B, C, ...
- Functions: $f(e_1, \ldots, e_n)$
- Definite descriptions: $\imath v{:}\, S$

Here v is a static variable, e_1,..., e_n are expressions and S is a formula. In the case of constants (nullary functions), we omit the parentheses.

2.1.3. Formulas

Below are permitted formulas:

- Predicates: $p(e_1, \ldots, e_n)$
- Equality: $e_1 = e_2$
- Logical connectives: $\neg S$, and $S \wedge T$
- Universal quantification: $\forall v{:}\, S$
- Unit interval: *skip*.
- Chop: $S;T$
- Chop-star: S^*, where S is a formula

Here v is any variable, e_1,..., e_n are expressions and S and T are arbitrary formulas.

We freely use various conventional propositional constructs which can be expressed in terms of $\wedge$ and $\neg$: *true*, *false*, $\vee$ (logical or), $\supset$ (implication), $\equiv$ (logical equivalence) and *if-then-else*. In addition, $\exists$ (existential quantification) is defined in terms of $\forall$. The following temporal operators are also quite standard:

$\bigcirc S$	$\stackrel{\text{def}}{=}$	$skip; S$	S is true from the next state
$\circledcirc S$	$\stackrel{\text{def}}{=}$	$\neg \bigcirc \neg S$	Weak next
$\Diamond S$	$\stackrel{\text{def}}{=}$	$true; S$	S is sometimes true
$\Box S$	$\stackrel{\text{def}}{=}$	$\neg \Diamond \neg S$	S is always true

Here is a version of $\bigcirc$ for expressions:

$$\bigcirc e \quad \stackrel{\text{def}}{=} \quad \imath a{:}\, \bigcirc(e = a),$$

where the static variable a has the same sort as e and does not occur freely in it.

We adapt the convention that S, T and U as well as primed and subscripted variants refer to arbitrary formulas. It is beneficial to sometimes consider formulas without any temporal modalities. We refer to these using w, w' and so forth. The symbols e, e' and so on refer to expressions. Expressions and modality-free formulas containing only static variables are themselves called static.

Formulas built from multiple occurrences of $\wedge$ are right-associative. Thus a formula of the form $S \wedge T \wedge U$ is equivalent to $S \wedge (T \wedge U)$. This also applies to formulas with *chop*, that is, a formula $S;T;U$ is equivalent to $S;(T;U)$. On several occasions, we write formula S^* using the alternative programming notation *for some times do* S.

2.1.4. Some sample formulas

Here are some sample ITL formulas whose semantics we will later describe:

$$\Box(I = 0); skip; \Box(I = 1), \qquad (skip; skip)^*, \qquad (skip; skip)^* \equiv (skip; skip)^{**},$$
$$I = 0 \wedge J = 0 \wedge (skip \wedge (\bigcirc I) = I + 1 \wedge (\bigcirc J) = J + I)^*,$$
$$I = 0 \wedge J = 0 \wedge (skip \wedge (\bigcirc I) = I + 1 \wedge (\bigcirc J) = J + I)^* \quad \supset \quad \Box(J = I(I-1) \div 2).$$

2.2. Semantics

2.2.1. Semantics of the underlying first-order logic

The semantics of ITL is built upon a fairly conventional first-order logic with sorts now described. We assume a fixed interpretation $\mathcal{I}$ which serves two purposes. First, it associates data domains $\mathcal{I}_1$, $\mathcal{I}_2$,..., with the corresponding sorts 1, 2,.... Secondly, $\mathcal{I}$ gives meaning to the predicate and function symbols. More precisely, $\mathcal{I}$ maps each n-ary predicate symbol p to an n-ary relation $\mathcal{I}(p) \in 2^{\mathcal{I}_{\hat{p}_1} \times \cdots \times \mathcal{I}_{\hat{p}_n}}$. Similarly, each n-ary function symbol f is associated with a n-ary function $\mathcal{I}(f) \in \mathcal{I}_{\hat{f}_1} \times \cdots \times \mathcal{I}_{\hat{f}_n} \to \mathcal{I}_{\hat{f}_{n+1}}$ that suits f's sort requirements. It is assumed that $\mathcal{I}$ contains interpretations for the arithmetic operators and relations for natural numbers as well as operators for manipulating finite lists (e. g., subscripting and list length).

The first-order logic uses a straightforward notion of *state*. A state is any function s which maps each variable v to a value $s(v)$ in the data domain $\mathcal{I}_{\hat{v}}$ indexed by v's sort $\hat{v}$. Unless we specify otherwise, variables in the range i, j, ..., n and I, J, ..., N are mapped to the natural numbers.

We assume the existence of a choice function χ which maps any nonempty set to some element in the set. This is needed for the semantics of definite descriptions.

2.2.2. Semantics of intervals

An interval is defined to be any finite, nonempty sequence of states such that every lower case, static variable is mapped to the same value in each state. The set of all intervals is denoted by *Int*. It is especially convenient to define the length of an interval σ, denoted $|\sigma|$, to be one less than the number of states in σ. Thus, the smallest interval has one state and length 0. The notation $\sigma_{i:j}$ denotes the subinterval of σ of length $j - i$ with states σ_i, σ_{i+1}, ..., σ_j. We write $\sigma \sim_v \sigma'$ if the intervals σ and σ' are identical with the possible exception of their mappings for the variable v.

2.2.3. Meaning of expressions

The meaning of an expression is defined inductively:

- Static or state variable: $\mathcal{M}_\sigma[\![v]\!] = \sigma_0(v)$.
  The value of a variable for an interval σ is the variable's value in the initial state σ_0.
- Function: $\mathcal{M}_\sigma[\![f(e_1, \ldots, e_n)]\!] = \mathcal{I}(f)(\mathcal{M}_\sigma[\![e_1]\!], \ldots, \mathcal{M}_\sigma[\![e_n]\!])$.
- Definite descriptions: $\mathcal{M}_\sigma[\![iv\colon S]\!] = \begin{cases} \chi(u) & \text{if } u \neq \{\,\} \\ \chi(\mathcal{I}_{\hat{v}}) & \text{otherwise,} \end{cases}$
  where u is the set of values of the static variable v in intervals σ' such that $\sigma \sim_v \sigma'$ and $\mathcal{M}_{\sigma'}[\![S]\!] = true$:

  $$u = \{\sigma'(v) : \sigma' \in Int,\ \sigma \sim_v \sigma' \text{ and } \mathcal{M}_{\sigma'}[\![S]\!] = true\}.$$

  If u is empty, the description equals some value selected from v's domain $\mathcal{I}_{\hat{v}}$. Since v is static, it has a unique value in σ' denoted here $\sigma'(v)$.

2.2.4. Meaning of formulas

- Predicates: $\mathcal{M}_\sigma[\![p(e_1,\ldots,e_n)]\!] = \mathit{true}$ iff $\langle \mathcal{M}_\sigma[\![e_1]\!],\ldots,\mathcal{M}_\sigma[\![e_n]\!]\rangle \in \mathcal{I}(p)$.
- Equality: $\mathcal{M}_\sigma[\![e_1 = e_2]\!] = \mathit{true}$ iff $\mathcal{M}_\sigma[\![e_1]\!] = \mathcal{M}_\sigma[\![e_2]\!]$.
- Negation: $\mathcal{M}_\sigma[\![\neg S]\!] = \mathit{true}$ iff $\mathcal{M}_\sigma[\![S]\!] = \mathit{false}$.
- Conjunction: $\mathcal{M}_\sigma[\![S \wedge T]\!] = \mathit{true}$ iff $\mathcal{M}_\sigma[\![S]\!] = \mathit{true}$ and $\mathcal{M}_\sigma[\![T]\!] = \mathit{true}$.
- Universal quantification: $\mathcal{M}_\sigma[\![\forall v\colon S]\!] = \mathit{true}$ iff $\mathcal{M}_{\sigma'}[\![S]\!] = \mathit{true}$, for all intervals σ' that are identical to σ except possibly for the behavior of the variable v (i. e., $\sigma \sim_v \sigma'$).
- Unit interval: $\mathcal{M}_\sigma[\![\mathit{skip}]\!] = \mathit{true}$ iff $|\sigma| = 1$.
- Chop: $\mathcal{M}_\sigma[\![S;T]\!] = \mathit{true}$ iff $\mathcal{M}_{\sigma'}[\![S]\!] = \mathit{true}$ and $\mathcal{M}_{\sigma''}[\![T]\!] = \mathit{true}$, where $\sigma' = \sigma_{0:k}$ and $\sigma'' = \sigma_{k:|\sigma|}$ for some $k \le |\sigma|$. Note that the two intervals σ' and σ'' share the common state σ_k.
- Chop-star: $\mathcal{M}_\sigma[\![S^*]\!] = \mathit{true}$ iff $\mathcal{M}_{\sigma_{l_i:l_{i+1}}}[\![S]\!] = \mathit{true}$, for each $i : 0 \le i < n$, for some $n \ge 0$ and finite sequence of one or more natural numbers $l_0 \le l_1 \le \cdots \le l_n$ where $l_0 = 0$ and $l_n = |\sigma|$. Note that S^* is true for any empty interval since we can always take $n = 0$.

2.2.5. Analysis of previous examples

Now that we have presented the basic semantics of ITL, it is possible to describe the meaning of the sample formulas given earlier. For each formula, we give the general characteristics of an interval on which the formula is true:

- $\Box(I = 0); \mathit{skip}; \Box(I = 1)$: The variable I equals 0 for a while and then equals 1 for the rest of the interval. We use *skip* to avoid $I = 0$ and $I = 1$ in one state.
- $(\mathit{skip}; \mathit{skip})^*$: The interval has an even length (i. e., an odd number of states).
- $(\mathit{skip}; \mathit{skip})^* \equiv (\mathit{skip}; \mathit{skip})^{**}$: The interval has even length iff it consists of a sequence of intervals each with even length.
- $I = 0 \wedge J = 0 \wedge (\mathit{skip} \wedge (\bigcirc I) = I + 1 \wedge (\bigcirc J) = J + I)^*$: The initial values of I and J are both 0 and then I increases from state to state by 1 while J simultaneously increases by I's value.
- $I = 0 \wedge J = 0 \wedge (\mathit{skip} \wedge (\bigcirc I) = I + 1 \wedge (\bigcirc J) = J + I)^* \supset \Box(J = I(I-1) \div 2)$: If within the interval the variables I and J have the behavior described in the previous example, then J always equals $I(I-1) \div 2$, that is, the sum of the numbers $0, 1, \ldots, I-1$.

2.2.6. Satisfiability, validity and provability

If a formula S is true for an interval σ, we say that σ *satisfies* S. A formula having some interval satisfying it is called *satisfiable*. A *valid* formula is one satisfied by all intervals. For example, the following two formulas are satisfiable but not valid:

$$I = 1; I = 2, \qquad \mathit{skip}.$$

Note that $I = 1$ and $I = 2$ are only examined in the initial states of the respective subintervals. Here are two formulas which are both satisfiable and valid:

$$(I = 1; I = 2) \equiv (I = 1 \wedge \Diamond(I = 2)), \qquad \mathit{skip}^*.$$

The formula skip^* is valid in part because *chop-star* is trivially true on empty intervals. The third and fifth examples given earlier in §2.1.4 are also valid whereas the others are

satisfiable but not valid.

Later on we present a proof system for ITL. From now one we prefix a formula with the conventional symbol $\vdash$ to indicate that the formula is provable. Much of the time, we give provable schemas representing classes of theorems. Here are some examples where S, S', T and U are arbitrary formulas and w is any formula containing no temporal operators:

$$\vdash \quad (S \vee S');T \quad \equiv \quad (S;T) \vee (S';T) \qquad \vdash \quad S^*;S^* \quad \supset \quad S^*$$
$$\vdash \quad (S;T) \wedge \neg(S;U) \quad \supset \quad S;(T \wedge \neg U) \qquad \vdash \quad S^{**} \quad \equiv \quad S^*$$
$$\vdash \quad (w \wedge S);T \quad \equiv \quad w \wedge (S;T)$$

Here are examples of provable theorems containing $\bigcirc$, $\Diamond$ and $\Box$:

$$\vdash \quad (\bigcirc S);T \equiv \bigcirc(S;T) \qquad \vdash \quad \Box(T \supset T') \wedge (S;T) \supset (S;T')$$
$$\vdash \quad S;T \supset \Diamond T \qquad \vdash \quad \bigcirc(e = e') \quad \supset \quad (\bigcirc e) = (\bigcirc e')$$

2.2.7. Complexity

If we restrict ourselves to ITL formulas containing only boolean-valued state and static variables, we have a propositional variant of the logic. Here is a useful result regarding this:

Theorem 2.1 (Halpern and Moszkowski) *Validity for quantifier-free ITL formulas containing only boolean-valued variables is decidable.*

See Moszkowski [2] for details. In general, complexity of the decision procedure is nonelementary. Kono [10] however has a useful implementation in Prolog.

2.3. Building a vocabulary

There are many kinds of dynamic phenomena that are useful to describe and reason about. The ITL formalism is powerful enough to express a rich class of concepts. Rather than limiting ourselves solely to formulas containing only the three basic ITL constructs, we develop what can be thought of as a vocabulary of various kinds of temporal behavior. This is an important task in its own right. An analogous situation arises in conventional propositional logic. One can start with one or two adequate connectives and then define a number of others. Once this is done, it is exceedingly unpleasant and unnatural to restrict propositional formulas and axiom systems to the original constructs. Regarding ITL, we similarly view the starting constructs as adequate connectives for defining a repertoire of others which are then freely used in formulas and in the proof system.

We now present a vocabulary of temporal concepts organized by categories. Propositional constructs are first looked at followed by first-order ones. After a group of definitions is given, a few representative theorems are shown. The reader may wish to try and understand their meaning and convince himself of their truth. They should not be thought of as arbitrary examples. Indeed, many of them have been extensively used by us in larger proofs. In additional, later definitions are often based on earlier definitions and therefore also serve as illustrations of their usage. The material here can be initially skimmed and then referred back to when the reader studies the examples of parallel systems discussed later in section 4.

2.3.1. Operators for initial and arbitrary subintervals

The conventional temporal operators $\bigcirc$, $\Diamond$ and $\Box$ defined earlier only look at subintervals that are suffixes of the current interval. We call these *terminal* subintervals. It is often

necessary to consider behavior in *initial* or *arbitrary* subintervals. The following operators serve this purpose:

$\diamondsuit_{\mathrm{i}} S \stackrel{\text{def}}{\equiv} S; true$	Some initial subinterval	$\diamondsuit_{\mathrm{a}} S \stackrel{\text{def}}{\equiv} true; S; true$	Some subinterval
$\boxdot_{\mathrm{i}} S \stackrel{\text{def}}{\equiv} \neg \diamondsuit_{\mathrm{i}} \neg S$	All initial subinbtervals	$\boxdot_{\mathrm{a}} S \stackrel{\text{def}}{\equiv} \neg \diamondsuit_{\mathrm{a}} \neg S$	All subintervals

The following are representative elementary theorems:

$$\vdash \quad \boxdot_{\mathrm{a}}(S \supset T) \wedge S^* \quad \supset \quad T^* \qquad\qquad \vdash \quad \boxdot_{\mathrm{a}} S \quad \equiv \quad \boxdot_{\mathrm{a}} \boxdot_{\mathrm{a}} S$$
$$\vdash \quad \boxdot_{\mathrm{i}}(S \supset S') \wedge (S;T) \quad \supset \quad (S';T) \qquad\qquad \vdash \quad \Box w \quad \equiv \quad \boxdot_{\mathrm{a}} \Box w$$

2.3.2. Interval length

The basic operator *skip* tests whether an interval has length 1, or in other words, exactly two states. We also need to be able to determine whether or not an interval has length 0 (i. e., exactly one state). Such intervals are called *empty*:

$$more \stackrel{\text{def}}{\equiv} \bigcirc true \quad \text{Nonempty interval} \qquad\qquad empty \stackrel{\text{def}}{\equiv} \neg more \quad \text{Empty interval}$$

Here are some provable properties of *empty* and *more*:

$$\vdash\ S; empty \equiv S \qquad\qquad \vdash\ more \equiv \Diamond skip \qquad\qquad \vdash\ empty\ \supset\ S^*$$

2.3.3. Final states

An interval's final state is identical to the terminal subinterval having length 0. We make use of this in the following definitions:

$$fin\ S \stackrel{\text{def}}{\equiv} \Box(empty\ \supset\ S) \quad \text{Final state} \qquad\qquad halt\ S \stackrel{\text{def}}{\equiv} \Box(empty \equiv S) \quad \text{Halt upon } S$$

The formula *fin* S only examines S's truth value in the final state whereas the formula *halt* S ensures that S is true exactly at the end and not before. The following theorems make use of these constructs:

$$\vdash \quad (S \wedge fin\ w); T \equiv S; (w \wedge T) \qquad\qquad \vdash \quad \neg S \wedge halt\ S\ \supset\ \bigcirc halt\ S$$
$$\vdash \quad S; (T \wedge fin\ w) \equiv (S;T) \wedge fin\ w \qquad\qquad \vdash \quad \Diamond w \equiv \diamondsuit_{\mathrm{i}}(halt\ w)$$

2.3.4. Unit subintervals

Our experience with very compositional properties has demonstrated the benefit of sometimes considering only subintervals having length 1. These subintervals are called *unit* subintervals. Here are constructs for examining them:

$$keep\ S \stackrel{\text{def}}{\equiv} \boxdot_{\mathrm{a}}(skip \supset S) \quad \text{All units} \qquad\qquad keepnow\ S \stackrel{\text{def}}{\equiv} \diamondsuit_{\mathrm{i}}(skip \wedge S) \quad \text{Initial unit}$$

The relation between *keep* and *keepnow* is similar to the one between $\Box$ and $\bigcirc$. For example, we use the construct *keepnow* in induction proofs involving *keep* . Here are some theorems:

$$\vdash \quad (keep\ S); (keep\ S)\ \supset\ keep\ S \qquad\qquad \vdash \quad (keepnow\ S) \wedge \bigcirc(keep\ S)\ \supset\ keep\ S,$$
$$\vdash \quad keep\ S \equiv keep\ keep\ S \qquad\qquad \vdash \quad keep\ w \equiv \Box(more \supset w),$$
$$\vdash \quad keep\ S \equiv (skip \wedge S)^* \qquad\qquad \vdash \quad \Box w \equiv keep\ w \wedge fin\ w.$$

2.3.5. Control and iterative constructs

It is well known that the conventional formula *if* S *then* T *else* U can be expressed in propositional logic as follows $(S \wedge T) \vee (\neg S \wedge U)$. The ITL formalism provides a natural way to define *if-then* and *while-do* as well. We also include here a very important analogue of Kleene plus for one or more iterations of a formula:

$$\begin{array}{lll} \mathit{if}\ S\ \mathit{then}\ T \stackrel{\text{def}}{\equiv} \mathit{if}\ S\ \mathit{then}\ T\ \mathit{else}\ \mathit{empty} & \text{If-then} & S^{+} \stackrel{\text{def}}{\equiv} S^{*}; S \quad \mathit{Chop\text{-}plus} \\ \mathit{while}\ S\ \mathit{do}\ T \stackrel{\text{def}}{\equiv} (S \wedge T)^{*} \wedge \mathit{fin}\ \neg S & \text{While loop} & \end{array}$$

The following are representative theorems:

$$\vdash \quad \mathit{while}\ w\ \mathit{do}\ S \quad \equiv \quad \mathit{if}\ w\ \mathit{then}\ (S; (\mathit{while}\ w\ \mathit{do}\ S)) \qquad \vdash \quad S^{+} \quad \equiv \quad S; S^{*}.$$
$$\vdash \quad \boxdot_{a}(S \supset T) \quad \supset \quad (\mathit{while}\ w\ \mathit{do}\ S) \supset (\mathit{while}\ w\ \mathit{do}\ T)$$

2.3.6. Some definite descriptions

We now look at some first-order ITL operators for expressions and formulas. Let us first consider some expressions definable using definite descriptions and operators already mentioned. Here e and e' are themselves arbitrary expressions of the same sort and a is any static variable not occurring freely in e and e' and having the same sort:

$$\mathit{fin}\ e \stackrel{\text{def}}{=} \imath a{:}\, \mathit{fin}\,(e = a) \qquad \mathit{keepnow}\ e \stackrel{\text{def}}{=} \imath a{:}\, \mathit{keepnow}\,(e = a)$$
$$\mathit{if}\ S\ \mathit{then}\ e\ \mathit{else}\ e' \stackrel{\text{def}}{=} \imath a{:}\,(\mathit{if}\ S\ \mathit{then}\ e = a\ \mathit{else}\ e' = a)$$

For example, *fin* e is the value of the expression e in the interval's final state. The restriction that the static variable a has the same sort as the expressions ensures that the descriptions also have this sort. The requirement in conditional expressions that e and e' have the same sort could be weakened but it suffices for our purposes. The reader probably finds the expression *keepnow* e unfamiliar. We use it when specifying and proving first-order properties of the *keep* construct.

Below are some theorems:

$$\vdash \quad S \supset (\mathit{if}\ S\ \mathit{then}\ e\ \mathit{else}\ e') = e \qquad \vdash \quad \mathit{more} \supset (\mathit{keepnow}\ \mathit{fin}\ A) = (\bigcirc A)$$
$$\vdash \quad \mathit{fin}\,(e = a) \equiv (\mathit{fin}\ e) = a$$

In the last theorem, A is an arbitrary state variable.

2.3.7. Some other first-order constructs

The next few definitions provide a way of observing the values of one or two expressions are various points in time. These constructs are used extensively in the parallel systems described later. As before, e and e' are arbitrary expressions of some particular sort and a is any static variable of the same sort not occurring freely in e:

$$\begin{array}{ll} e \leftarrow e' \stackrel{\text{def}}{\equiv} (\mathit{fin}\ e) = e' & \mathit{goodindex}\ e \stackrel{\text{def}}{\equiv} \mathit{keep}\,(e \leq \bigcirc e \leq e + 1) \\ \mathit{stable}\ e \stackrel{\text{def}}{\equiv} \exists a{:}\, \Box(e = a) & \mathit{padded}\ e \stackrel{\text{def}}{\equiv} \exists a{:}\, \mathit{keep}\,(e = a) \\ e\ \mathit{gets}\ e' \stackrel{\text{def}}{\equiv} \mathit{keep}\,(e \leftarrow e') & e \lessapprox e' \stackrel{\text{def}}{\equiv} (e \leftarrow e') \wedge \mathit{padded}\ e \end{array}$$

The operator *goodindex* tests that an expression remains unchanged or increases by 1 over every unit subinterval. The operator *padded* ensures that the expression's value remains

unchanged except possibly at the interval's very end. This is used in the definition of the operator $\lessapprox$ to describe a *padded temporal assignment* which can be used for synchronous assignments in parallel systems.

We now give some sample theorems. The variables A, K and L are state variables and n is a static one with K, L and n ranging over the natural numbers. We assume that w is a state formula in which A is the only state variable occurring.

$$\begin{array}{llll}
\vdash & w^e_A \wedge A \leftarrow e \supset \mathit{fin}\ w & \vdash & \mathit{stable}\ A \equiv (\mathit{stable}\ A)^*, \\
\vdash & K \leftarrow K \supset \neg(K \leftarrow K+1) & \vdash & \mathit{stable}\ A \equiv A\ \mathit{gets}\ A, \\
\vdash & K+n \leftarrow L+n \equiv K \leftarrow L & \vdash & \mathit{stable}\ A \supset A \leftarrow A \\
\vdash & K \leq L \wedge \mathit{goodindex}\ L \wedge \mathit{stable}\ K \supset \Box(K \leq L) & & \\
\vdash & K \lessapprox K+1 \equiv \mathit{stable}\ K; (\mathit{skip} \wedge K \leftarrow K+1) & &
\end{array}$$

3. A PRACTICAL PROOF SYSTEM

We now present a very powerful and practical compositional proof system for ITL. Our experience in rigorously developing hundreds of propositional and first-order proofs has helped us refine the axioms and convinced us they are sufficient for a very wide range of purposes. The proof system is divided into a propositional part and a first-order part. Our discussion looks at each in turn.

3.1. Propositional axioms and inference rules

The propositional axioms and inference rules mainly deal with *chop*, and *skip* and operators derived from them. Only one axiom is needed for *chop-star.* The proof system gives nearly equal treatment to initial and terminal subintervals. This is exceedingly important for the kinds of proofs we do. In addition, this makes the proof system easier to understand since much of it consists simply of duals in this sense. In contrast, most temporal logics cannot handle initial subintervals and even other proof systems for ITL largely neglect them.

Rosner and Pnueli [11] and Paech [12] give propositional proof systems for ITL with infinite intervals and prove completeness. Our proof system contains some of the propositional axioms suggested by Rosner and Pnueli but also includes our own axioms and inference rule for the operators $\boxdot$, *keepnow* , and *chop-star*. These assist in deducing propositional and first-order theorems and in deriving rules for importing, exporting and other aspects of composition.

Prop $\vdash$ Substitutions of tautologies
P2 $\vdash (S;T);U \equiv S;(T;U)$
P3 $\vdash (S \vee S');T \supset (S;T) \vee (S';T)$
P4 $\vdash S;(T \vee T') \supset (S;T) \vee (S;T')$
P5 $\vdash \mathit{empty};S \equiv S$
P6 $\vdash S;\mathit{empty} \equiv S$
P7 $\vdash w \supset \boxdot w$

P8 $\vdash \boxdot(S \supset S') \wedge \Box(T \supset T') \supset (S;T) \supset (S';T')$
P9 $\vdash \bigcirc S \supset \neg \bigcirc \neg S$
P10 $\vdash \mathit{keepnow}\ S \supset \neg\, \mathit{keepnow}\ \neg S$
P11 $\vdash S \wedge \Box(S \supset \circledcirc S) \supset \Box S$
P12 $\vdash S^* \equiv \mathit{empty} \vee (S \wedge \mathit{more});S^*$

MP $\vdash S \supset T,\ \vdash S \Rightarrow \vdash T$
$\Box$Gen $\vdash S \Rightarrow \vdash \Box S$
$\boxdot$Gen $\vdash S \Rightarrow \vdash \boxdot S$

We now give a sample theorem and its proof:

$$\vdash \quad \boxdot(S \supset T) \quad \supset \quad \diamondsuit S \supset \diamondsuit T$$

Proof:

1	$\vdash$	$true \quad \supset \quad true$	**Prop**
2	$\vdash$	$\Box(true \supset true)$	1,$\Box$**Gen**
3	$\vdash$	$\boxdot(S \supset T) \wedge \Box(true \supset true) \quad \supset \quad (S; true) \supset (T; true)$	**P8**
4	$\vdash$	$\boxdot(S \supset T) \quad \supset \quad (S; true) \supset (T; true)$	2,3,**Prop**
5	$\vdash$	$\boxdot(S \supset T) \quad \supset \quad \diamondsuit S \supset \diamondsuit T$	4,def. of $\diamondsuit$

Theorem 3.1 *The propositional proof system is complete for quantifier-free formulas containing only boolean-valued static and state variables.*

Outline of proof: For a given formula, we construct a finite tableau consisting of a number of states. Each state is represented as a disjunction whose disjuncts are themselves conjunctions of primitive propositions, *next* formulas and their negations. Now suppose S is a valid formula. Construct a tableau for its negation $\neg S$. Call a state in a tableau *final* if it is satisfiable by some empty interval. No state reachable from the initial state in our tableau for $\neg S$ is final, since otherwise we can use the path to construct a model for $\neg S$. Therefore the tableau reflects that $\neg S$ is not true in any finite intervals. We convert this to a proof-by-contradiction for S. This technique also applies to a version of Rosner and Pnueli's proof system restricted to finite intervals.

3.2. First-order axioms and inference rules

Below are axioms and inference rules for reasoning about first-order concepts. They are to be used together with the propositional ones already introduced. See Manna [13] and Kröger [14] for proof systems for *chop*-free first-order temporal logic. We let v and v' refer to both static and state variables.

F1 $\vdash$ All substitution instances of valid nonmodal formulas of conventional first-order logic with arithmetic

F2 $\vdash \forall v{:}\, S \quad \supset \quad S^e_v$,
where the expression e is compatible with v and v is free for e in S.

F3 $\vdash \forall v{:}\,(S \supset T) \supset (S \supset \forall v{:}\, T)$, where v doesn't occur freely in S.

F4 $\vdash (\imath v{:}\, S) = (\imath v'{:}\, S^{v'}_v)$,
where v and v' are static variables of one sort and v is free for v' in S.

F5 $\vdash \forall v{:}\,(S \equiv T) \quad \supset \quad (\imath v{:}\, S) = (\imath v{:}\, T)$, where v is static.

F6 $\vdash (\exists v{:}\, S) \wedge (\imath v{:}\, S) = v \quad \supset \quad S$, where v is a static variable.

F7 $\vdash w \quad \supset \quad \Box w$, where w only contains static variables.

F8 $\vdash \exists v{:}\,(S;T) \quad \supset \quad (\exists v{:}\, S); T$, where v doesn't occur freely in T.

F9 $\vdash \exists v{:}\,(S;T) \quad \supset \quad S; (\exists v{:}\, T)$, where v doesn't occur freely in S.

F10 $\vdash (\exists v{:}\, S); \bigcirc(\exists v{:}\, T) \supset \exists v{:}\,(S; \bigcirc T)$, where v is a state variable.

$\forall$**Gen** $\vdash S \quad \Rightarrow \quad \vdash \forall v{:}\, S,$ for any variable v.
Induct $\vdash S_n^0, \quad \vdash S \supset S_n^{n+1} \quad \Rightarrow \quad \vdash S,$
for any static variable n whose sort is the natural numbers.

The axiom **F1** permits using properties of conventional first-order logic with arithmetic without proof. Most of the other axioms and the two inference rules at the end are straightforward adaptations of conventional nonmodal equivalents for quantifiers and definite descriptions. Only four axioms actually contain temporal operators. Axiom **F7** deals with state formulas containing only static variables. The two axioms **F8** and **F9** show how to move an existential quantifier out of the scope of *chop*. The remaining temporal axiom **F10** shows how to combine two state variables in nearly adjacent subintervals into one state variable for the entire interval. We extensively use it and lemmas derived from it for constructing auxiliary variables.

3.3. ITL with infinite time

The semantics and proof system so far presented is suitable for reasoning about finite intervals. We briefly discuss some modifications needed to permit infinite intervals as well. First, we apply our semantics of $S;T$ and S^* to infinite intervals. As before this means $S;T$ is true if the interval can be divided into one part for S and another adjacent part for T and that S^* is true if the interval can be divided into a finite number of parts, each satisfying S. In addition, we now let $S;T$ be true on an infinite interval which satisfies S. For such an interval, we can ignore T. Furthermore, we let S^* be true on an infinite interval dividable into an infinite number of finite intervals each satisfying S. We define new constructs for testing whether an interval is infinite or finite, and alter the definition of $\Diamond$:

$$\begin{array}{lll} \mathit{inf} & \stackrel{\text{def}}{\equiv} & \mathit{true};\mathit{false} \\ \Diamond S & \stackrel{\text{def}}{\equiv} & \mathit{finite};S \end{array} \qquad\qquad \mathit{finite} \stackrel{\text{def}}{\equiv} \neg\mathit{inf}$$

Once this is done, all the axioms and basic inference rules remain sound. We also include the following two axioms:

P13 $\vdash (S \wedge \mathit{inf});T \equiv S \wedge \mathit{inf}$
P14 $\vdash S \wedge \Box(S \supset (T \wedge \mathit{more});S) \supset T^*$

It seems likely that completeness in the sense of theorem 3.1 can only be achieved with a nonconventional inference rule. This is not central to our approach. In the rest of this paper we restrict ourselves to finite time.

4. APPLICATIONS

Let us now consider some applications of the proof system. Due to space limitations and for the sake of brevity, only summaries of the actual proofs are given here. The reader may wish to review the material in the introduction about very compositional properties.

4.1. A simple parallel system with shared write access

Below are descriptions of two simple processes Q and R which alternately modify a single variable K:

$$
\begin{array}{ll}
Q(K) \stackrel{\text{def}}{\equiv} & R(K) \stackrel{\text{def}}{\equiv} \\
\quad \textit{for some times do} \; (& \quad \textit{for some times do} \; (\\
\quad\quad K \lessapprox K+1; & \quad\quad \textit{halt odd}(K); \\
\quad\quad \textit{halt even}(K) & \quad\quad K \lessapprox K+1 \\
\quad) & \quad)
\end{array}
$$

The iterating in Q and R is expressed by means of the *chop-star* operator in the notation of a *for-loop*. The predicates *even* and *odd* are simple arithmetic tests. Here is the overall system together with K initially equal to 0:

$$K = 0 \wedge Q(K) \wedge R(K).$$

When K is even, Q keeps it stable for a while and then eventually increments it, thus making it odd. At this time, R keeps K stable and then increments it, thus handing responsibility for it back to Q. This continues for some finite, unspecified number of times. We use padded temporal assignments in order to ensure proper communication between Q and R.

Here is a theorem describing correctness of the overall system:

$$\vdash \quad \textit{even}(K) \wedge Q(K) \wedge R(K) \quad \supset \quad \textit{goodindex}\ K \wedge \textit{fin even}(K).$$

The theorem uses the *goodindex* operator defined earlier to state that K is always stable or increases by 1. In addition, K's final value is even.

In the proof we make use of the following variants of *goodindex* :

$$
\begin{array}{lll}
\textit{goodevenindex}\,(K) & \stackrel{\text{def}}{\equiv} & \textit{keep}\,(\textit{even}(K) \ \supset \ K \leq \bigcirc K \leq K+1), \\
\textit{goododdindex}\,(K) & \stackrel{\text{def}}{\equiv} & \textit{keep}\,(\textit{odd}(K) \ \supset \ K \leq \bigcirc K \leq K+1).
\end{array}
$$

Both *goodevenindex* and *goododdindex* are fixed points of *keep* so we can readily prove the following lemmas for Q and R using sequential composition:

$$
\begin{array}{lll}
\vdash & \textit{even}(K) \wedge Q(K) & \supset \quad \textit{goodevenindex}\,(K) \wedge \textit{fin even}(K), \\
\vdash & \textit{even}(K) \wedge R(K) & \supset \quad \textit{goododdindex}\,(K) \wedge \textit{fin even}(K).
\end{array}
$$

The formulas $\textit{goodevenindex}\,(K)$ and $\textit{goododdindex}\,(K)$ can then be combined together into $\textit{goodindex}\,(K)$:

$$\vdash \quad \textit{goodevenindex}\,(K) \wedge \textit{goododdindex}\,(K) \quad \supset \quad \textit{goodindex}\,(K).$$

The proofs of all these theorems makes extensive use of the fact that *goodindex* and its variants are very compositional.

4.2. Skeletons and compositionality

One of the simplest types of sequential communicating processes are those expressible in the form S^*, where the arbitrary formula S describes some basic transaction. Such formulas are clearly fixpoints of *chop-star*. The Q and R formulas given earlier are examples.

A number of communication techniques do not merely consist of a sequential series of similar transactions, one following the other. Instead, between each pair of consecutive transactions there can be an idle period described by some formula T. It is useful to restrict T to a fixpoint of *chop-star* since this ensures that any empty interval is a (trivial) idle period and that adjacent idle periods can be lumped together and thought of as one. The simplest kind of idling involves one or more state variables being kept stable but other forms of idling are also possible.

Once we introduce the notion of idling, the skeletal behavior to be extracted from an appropriate specification can be roughly described as something of the general form

$$T; S; T; S; \ldots; S; T.$$

Here each S corresponds to an individual transaction and each T to an idle period. This can be more precisely expressed in closed form as $(T; S)^*; T$. It can be proved that such a formula is a fixpoint of *chop-star* and can therefore be compositionally extracted from specifications.

For succinctness, let us further require that the transaction S absorbs T on the left:

$$\vdash \quad T; S \quad \supset \quad S.$$

If this is not provable for a particular S, one can simply use a new transaction S' defined as $T; S$ which is easily seen to have the desired property. If S and T are as described, we call the formula $S^*; T$ an *S-star-T* formula. Such a formula is then a very convenient fixpoint of *chop-star*. The formulas S, T and $(S^*; T)^*$ each imply $S^*; T$. It can be modularly exported from applications and then analyzed. The results can later be imported into proofs specific to the applications.

4.3. Producer-consumer system with single buffer

Consider the following skeletal producer-consumer system which illustrates a general message-passing convention using a single buffer:

$$\begin{array}{l} \mathit{BufProdSkel}(\mathit{CI}, \mathit{PI}, \mathit{Buf}) \stackrel{\text{def}}{\equiv} \\ \quad \mathit{for\ some\ times\ do}\ (\\ \quad\quad \mathit{PI} \mathrel{\ll\!\sim} \mathit{PI} + 1; \\ \quad\quad \mathit{stable}\ (\mathit{PI}, \mathit{Buf}) \wedge \mathit{fin}\ (\mathit{PI} \neq \mathit{CI} + 1) \\ \quad); \\ \quad \mathit{stable}\ \mathit{PI} \end{array} \qquad \begin{array}{l} \mathit{BufConSkel}(\mathit{CI}, \mathit{PI}) \stackrel{\text{def}}{\equiv} \\ \quad \mathit{for\ some\ times\ do}\ (\\ \quad\quad \mathit{stable}\ \mathit{CI} \wedge \mathit{fin}\ (\mathit{CI} \neq \mathit{PI}); \\ \quad\quad \mathit{CI} \mathrel{\ll\!\sim} \mathit{CI} + 1 \\ \quad); \\ \quad \mathit{stable}\ \mathit{CI} \end{array}$$

Recall that we use a for-loop notation as a programming-language syntax for *chop-star*. The body of each skeleton's loop contains one transaction and the last subformula after the loop describes the idle behavior. The producer is responsible for an index PI and a buffer Buf and the consumer maintains a similar index CI.

It should be easy to see that both skeletons meet the requirements of *S-star-T* formulas and can therefore be compositionally extracted from systems using this message-passing convention. The idle periods represented by *stable* are readily absorbed on the left of the respective transactions which themselves start with stable periods. Two stable periods can be merged into one.

The combined system skeleton *BufSysSkel* is simply the conjunction of the producer and consumer skeletons:

$$BufSysSkel(CI, PI, Buf) \quad \stackrel{\text{def}}{\equiv} \quad BufProdSkel(CI, PI, Buf) \wedge BufConSkel(CI, PI).$$

Note that *BufSysSkel* is not a fixpoint of chop-star and therefore lacks sequential compositionality. This is not a problem for us since we only use *BufSysSkel* after we have extracted the individual producer and consumer skeletons.

So far we have discussed how to obtain the skeletons from specifications of applications. Another important issue is how to describe the correctness of the skeleton system. In order to do this, we first formalize a notion of processing a sequence of data values. The following definition for *goodbuffer* serves this purpose:

$$goodbuffer\,(tr, K, A) \quad \stackrel{\text{def}}{\equiv} \quad keep\,(K \leftarrow K + 1 \;\supset\; fin\,(K \leq |tr| \wedge A = tr[K-1])).$$

Here tr is a static, possibly empty list which traces the data values being transmitted. The index variable K is a state variable ranging over the natural numbers which increases by 1 whenever a new data value is seen. The definition ensures that when this happens, K does not exceed the length of the trace tr. The variable A serves as a buffer containing that current data value. Whenever a new data value is handled, A should contain the value in the trace tr indexed by $K-1$. Note that the trace can include elements not indexed by K in the current interval. Being a fixpoint of *keep* , *goodbuffer* is easy to import and export within specifications.

The following theorem characterizes the basic correctness of the skeleton *BufSysSkel*:

$$\begin{array}{lll} \vdash & CI = PI \wedge goodbuffer\,(tr, PI, Buf) \wedge BufSysSkel(PI, CI, Buf) \\ & \supset & goodbuffer\,(tr, CI, Buf) \wedge fin\,(CI = PI). \end{array}$$

We also need the following lemmas in order to verify that the producer and consumer are patient:

$$\begin{array}{lll} \vdash & CI = PI \wedge BufSysSkel(PI, CI, Buf) & \supset \quad keep\,(CI = PI \;\supset\; stable\;\; CI), \\ \vdash & CI = PI \wedge BufSysSkel(PI, CI, Buf) & \supset \quad keep\,(PI = CI + 1 \;\supset\; stable\,(PI, Buf)). \end{array}$$

The first lemma ensures that the consumer patiently waits for the producer and does not increase its index before the producer makes data available. The second lemma similarly ensures that the producer index and buffer remain stable until after the consumer has indicated acceptance. From these lemmas, we can prove that the producer and consumer indices never differ by much:

$$\vdash \quad CI = PI \wedge BufSysSkel(PI, CI, Buf) \quad \supset \quad \Box(CI \leq PI \leq CI + 1).$$

The proofs of these properties make extensive use of compositionality and various other properties of *goodindex* , *goodbuffer* and other constructs introduced so far. For example, when an index is stable and no data is therefore communicated, *goodbuffer* is trivially true:

$$\vdash \quad \mathit{stable}\ K \quad \supset \quad \mathit{goodbuffer}\,(tr, K, A).$$

The following theorem ensures that we can always assume the existence of a trace associated with a given well-behaved index and buffer:

$$\vdash \quad \mathit{goodindex}\ K \quad \supset \quad \exists tr\colon (\mathit{goodbuffer}\,(tr, K, A) \wedge \mathit{fin}\,(K = |tr|)).$$

4.4. An asynchronous system

Our previous example deals with data transfers using a single buffer. This approach requires the producer to wait for consumer acknowledgement every time data is made available. Now consider a system with a pool of n buffers, for some $n \geq 1$. The producer can advance several data transfers ahead of the consumer. We now look at skeletons for such behavior. For simplicity, we only consider the interaction of the producer and consumer indices and omit the buffer pool.

$$\begin{array}{l} \mathit{IndexProdSkel}(n, PI, CI) \stackrel{\text{def}}{\equiv} \\ \quad \mathit{for\ some\ times\ do}\ (\\ \quad\quad PI \ll\sim PI + 1; \\ \quad\quad \mathit{stable}\ PI \wedge \mathit{fin}\,(PI \neq CI + n) \\ \quad); \\ \quad \mathit{stable}\ PI \end{array} \qquad \begin{array}{l} \mathit{IndexConSkel}(PI, CI) \stackrel{\text{def}}{\equiv} \\ \quad \mathit{for\ some\ times\ do}\ (\\ \quad\quad \mathit{stable}\ CI \wedge \mathit{fin}\,(CI \neq PI); \\ \quad\quad CI \ll\sim CI + 1 \\ \quad); \\ \quad \mathit{stable}\ CI \end{array}$$

The static variable n equals the number of buffer slots, PI is the producer buffer index and CI is the consumer buffer index. At any point in time, the values of PI mod n and CI mod n refer to the respective current buffer slots. The skeleton $\mathit{IndexProdSkel}(n, PI, CI)$ controls the behavior of PI and similarly $\mathit{IndexConSkel}(PI, CI)$ looks after CI. Both skeletons are *S-star-T* formulas.

Here is the combined system:

$$\mathit{IndexSysSkel}(n, PI, CI) \quad \stackrel{\text{def}}{\equiv} \quad \mathit{IndexProdSkel}(n, PI, CI) \wedge \mathit{IndexConSkel}(PI, CI).$$

The following two theorems show that the producer waits if it is n transfers ahead of of the consumer and that the consumer waits until the producer is ahead of it:

$$\begin{array}{ll} \vdash & CI = PI \wedge \mathit{IndexSysSkel}(n, PI, CI) \\ & \quad \supset \quad \mathit{keep}\,(PI = CI + n \ \supset\ \mathit{stable}\ PI) \wedge \mathit{fin}\,(PI < CI + n), \\ \vdash & CI = PI \wedge \mathit{IndexSysSkel}(n, PI, CI) \\ & \quad \supset \quad \mathit{keep}\,(CI = PI \ \supset\ \mathit{stable}\ CI) \wedge \mathit{fin}\,(CI \leq PI). \end{array}$$

The proofs use reduced forms of the skeletons not given here. From these theorems we can also prove that PI always ranges between CI and $CI + n$:

$$\vdash \quad CI = PI \wedge \mathit{IndexSysSkel}(n, PI, CI) \quad \supset \quad \Box(CI \leq PI \leq CI + n).$$

4.5. Specification of timing constraints

The ITL formalism provides facilities for specifying and proving theorems involving timing dependencies. We now introduce constructs to measure interval length and then show an application using them. The following definition $intlen\,(n)$ can be used to determine whether an interval has length equal to the static variable n:

$$intlen\,(n) \quad \stackrel{\text{def}}{\equiv} \quad \exists I\colon (I = 0 \wedge I \; gets \; I + 1 \wedge fin\,(I = n)).$$

This existentially introduces a state variable I which counts the number of units of time in the interval. The interval has length n exactly if I's final value equals n. Within ITL we can prove that every interval has some length and that the length is unique:

$$\vdash \quad \exists n\colon intlen\,(n) \qquad \vdash \quad intlen\,(m) \wedge intlen\,(n) \quad \supset \quad m = n$$

The existence and uniqueness of interval length helps us define a definite description equaling the interval's length:

$$len \quad \stackrel{\text{def}}{=} \quad \imath n\colon intlen\,(n).$$

The expression len has the following provable theorem for additivity of interval lengths:

$$\vdash \quad len\; = m + n \quad \equiv \quad len\; = m; len\; = n.$$

We now specify skeletons for a simple timing-dependent system with a producer and a consumer: Here are skeletons:

$$\begin{array}{l} TimedProdSkel(n, PI, Buf) \quad \stackrel{\text{def}}{\equiv} \\ \quad for\ some\ times\ do\ (\\ \quad\quad stable\ PI; \\ \quad\quad PI \lessapprox PI + 1; \\ \quad\quad len\; \geq n \wedge stable\,(PI, Buf) \\ \quad); \\ \quad stable\ PI \end{array} \qquad \begin{array}{l} TimedConSkel(n, PI, CI) \quad \stackrel{\text{def}}{\equiv} \\ \quad for\ some\ times\ do\ (\\ \quad\quad halt\,(CI \neq PI) \wedge stable\ CI; \\ \quad\quad len\; \leq n \wedge CI \lessapprox CI + 1 \\ \quad); \\ \quad stable\ CI \end{array}$$

As before, the producer skeleton uses the index PI to indicate when a new data value is ready in the buffer Buf. Unlike our earlier examples, the producer does not examine the consumer index CI to wait for receipt. Instead, the producer keeps the data stable for a minimum of n units of time before making new data available. The consumer skeleton waits for the producer prepare a value and then accepts it in not more than n units of time. Note that the producer skeleton is an *S-star-T* formula but the consumer skeleton is not since extra idling can adversely affect its timely response to incoming data.

As in previous examples, we specify the combined system skeleton using conjunction:

$$\begin{array}{l} TimedSysSkel(n, PI, CI, Buf) \stackrel{\text{def}}{\equiv} \\ \quad TimedProdSkel(n, PI, Buf) \wedge TimedConSkel(n, PI, CI). \end{array}$$

The basic correctness for data transfer is expressed by the following theorem:

$$\begin{array}{ll} \vdash & CI = PI \wedge goodbuffer\,(tr, PI, Buf) \wedge TimedSysSkel(n, PI, CI, Buf) \\ \quad \supset & goodbuffer\,(tr, CI, Buf) \wedge fin\,(CI \leq PI). \end{array}$$

Here the final value of CI is not necessarily equal to PI. This is because the producer does not explicitly wait for the consumer and can produce more data than the consumer is interested in seeing.

The proof of correctness uses a special timer counter Ti which is existentially introduced without loss of generality as an auxiliary variable to keep track of how much time has elapsed since the producer index PI last changed. We define that relationship between Ti and an arbitrary state variable A (such as PI in our case) as follows:

$$goodtimer\,(Ti, A) \quad \stackrel{\text{def}}{\equiv} \quad Ti\ gets\ (if\ stable\ A\ then\ Ti + 1\ else\ 0).$$

Thus, Ti always increases by 1 except when A changes and resets Ti to 0. In our example here, the producer is verified to keep PI and Buf stable as long as Ti is less than n:

$$\begin{array}{ll} \vdash & Ti \geq n \wedge goodtimer\,(Ti, PI) \wedge TimedSysSkel(n, PI, CI, Buf) \\ & \quad \supset \quad keep\,(Ti < n \ \supset \ stable\,(PI, Buf)) \wedge fin\,(Ti \geq n). \end{array}$$

For simplicity we assume that Ti is initially not less than n since this mean PI can change at any time.

As mentioned above, the auxiliary variable Ti can be introduced without loss of generality. The following shows this:

$$\vdash \quad \exists Ti\colon (Ti = n \wedge Ti\ gets\ (if\ stable\ PI\ then\ Ti + 1\ else\ 0)).$$

This is in fact a straightforward corollary of a powerful and more general theorem we prove about constructing auxiliary variables:

$$\vdash \quad \exists A\colon (A = e \wedge A\ gets\ e').$$

Here the state variable A and expressions e and e' have the same sort. There should be no free occurrences at all of A in e. We allow A to occur in e' but not within the context of temporal operators. These restrictions ensure that A is not circularly defined. For example, the following simple corollary constructs a counter that can be used for showing the existence of interval length needed for the constructs $intlen$ and len :

$$\vdash \quad \exists I\colon (I = 0 \wedge I\ gets\ I + 1).$$

5. CONCLUSION

Interval Temporal Logic is a simple and yet powerful formalism for dealing with dynamic behavior. From only a few basic concepts and a reasonably small axiom system, a wealth of useful operators and very compositional properties concerning sequential and parallel activity can be investigated. Facilities exist for dealing with behavior involving message passing, shared write access and timing constraints. Until now, little work has been directed at doing formal proofs about applications directly in ITL. We have presented some results of our extensive experience based on hundreds of proofs and believe that this approach is very promising. It may also be of benefit in proof systems for the Duration Calculus [15], a continuous-time variant of ITL.

Acknowledgements

We wish to thank Zhenhua Duan, John Fitzgerald, Roger Hale, Chris Holt, Shinji Kono, Maciej Koutny, Anders Ravn and Hussein Zedan for discussions. The Science and Engineering Research Council funded our research.

REFERENCES

1. C. A. R. Hoare. An axiomatic basis for computer programming, Comm. ACM, Vol. 12, No. 10, 1969, 576–580, 583.
2. B. Moszkowski. Reasoning about Digital Circuits, PhD thesis, Stanford Univ., 1983.
3. J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals, in: ICALP83, LNCS 154, Springer-Verlag, Berlin, 1983, 278-291.
4. B. Moszkowski. A temporal logic for multilevel reasoning about hardware, IEEE Computer, Vol. 18, No. 2, 1985, 10–19.
5. B. Moszkowski. Executing Temporal Logic Programs, Cambridge Univ. Press, Cambridge, UK, 1986.
6. C. Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language, Theoret. Comp. Sci., Vol. 58, 1988, 347–359.
7. C. B. Jones. Specification and design of (parallel) programs, in: Proc. Information Processing '83, R. E. A. Mason (ed.), Elsevier, Amsterdam, 1983, 321–332.
8. N. Francez and A. Pnueli. A proof method for cyclic programs, Acta Informatica, Vol. 9, 1978, 133–157.
9. P. Cousot. Methods and Logics for Proving Programs, in: Handbook of Theoret. Comp. Sci. (Vol. B), J. van Leeuwen (ed.), Elsevier, Amsterdam, 1990, 841–993.
10. S. Kono. Automatic verification of Interval Temporal Logic, Tech. rep. SCSL-TM-92-007, Sony Computer Sci. Lab., Inc., Tokyo, Japan, 1993.
11. R. Rosner and A. Pnueli. A choppy logic, in Proc. 1st Ann. IEEE Symp. on Logic In Comp. Sci., IEEE, 1986, 306–314.
12. B. Paech. Gentzen-systems for propositional temporal logics, in: Proc. 2nd Workshop on Comp. Sci. Logic, LNCS 385, Springer-Verlag, Berlin, 1988, 240–253.
13. Z. Manna. Verification of sequential programs: Temporal axiomatization, in: M. Broy and G. Schmidt (eds.), Theoretical Foundations of Programming Methodology, D. Reidel Pub. Co., 1982, 53–102.
14. F. Kröger. Temporal Logic of Programs, Springer-Verlag, Berlin, 1987.
15. Zhou Chaochen, C. A. R. Hoare and A. P. Ravn. A calculus of durations, Inf. Proc. Let., Vol. 40, No. 5, 1991, 269–276.