Research Report 203

Sometimes 'Some' is as Good as 'All'
(Preliminary Version)

Doron Peled

RR203

The representation of partial order semantics as an equivalence relation on interleaving sequences extends the expressive power of interleaving semantics. A specification formalism called *existential specification* is introduced: a formula is interpreted over equivalence classes of sequences by asserting that *some* (at least one but not necessarily all) sequences from each equivalence class satisfy a given property. It differs from the more common *universal specification*, which is interpreted over *all* sequences in all classes. Its advantage over other formalisms that deal with partial order executions lies in its simplicity: any syntax that is defined over interleaving sequences, e.g., linear temporal logic, can be adopted. It is shown how under an appropriate semantical construction, an exact existential specification of a program (i.e., each property of the program expressed using the formalism is a consequence of this specification) can be given. Moreover, under such a construction, no information about the program is lost by choosing exact existential specification rather than exact universal specification; it is possible to generalise by means of a proof rule an exact existential specification into an exact universal specification. Deducing an exact existential specification from an exact universal specification is also possible. This provides a relatively complete proof system for existential specification. Applications of these results to compositionality are discussed.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

March 1992

1 Introduction

There has been a long lasting debate about the use of partial order semantics versus using interleaving semantics to model concurrent programs. Supporters of interleaving semantics often claim that this model includes all the information needed to be expressed about a program, and that partial order semantics adds no important extra information. Supporters of the partial order model show how certain properties are better described and proved using partial order semantics.

Representing partial order execution as an equivalence class of interleaving sequences (or equivalently, traces) provides a way to connect the two models. Using this model, there is a common paradigm of properties that requires that *at least one interleaving sequence from each equivalence class satisfies some conditions*. We shall call this *existential specification*, as opposed to the more common *universal specification*. Using existential specification is sometimes more convenient than asserting over *all* the interleaving sequences, as some interleaving sequences represent the desired property better than others.

One example is serializability of database programs [23], where each execution sequence is supposed to be equivalent to a sequence in which the transitions are executed one after the other. Other examples include specifying algorithms such as Chandy and Lamport's global snapshot algorithm [5], recovery from faults based on this algorithm [24], and layering of concurrent programs [8]. Program verification methods can exploit this paradigm by allowing proofs of the properties of programs using intermediate assertions that apply only to the states of these representatives [16, 29], while in model checking it is used to reduce the size of the checked structure [30, 11]. Since existential specification covers a wide range of properties and can use a simple formalism such as linear temporal logic (LTL), it can be considered as an alternative to more powerful specification languages for partial order semantics [27, 15, 26].

Expressing properties of representatives rather than all the execution sequences was recognised in [15]. There, it was described as a class of properties embedded in the more powerful logic **ISTL***. In this paper, an alternative representation of this class of properties using a simpler framework that uses the syntax of LTL is suggested, and is further studied.

This paper deals with some basic questions about representing properties using existential specification, and its interrelation with the more familiar universal specification. The ability to specify *exactly* all the executions of a given program using existential specification is studied. Such a specification is desirable as all the properties of the program that are expressible in this formalism are consequences of it. An exact specification is often used for completeness proofs [19] and is especially important for compositional proof methods [4]. It is shown that provided that the semantical construction satisfy certain requirements, an exact existential specification can be given.

We prove that it is possible to conclude from an existential specification properties of *all* the sequences: this arises when one wants to *generalise* or conclude from an existential property a universal property of the program. It is shown that under an appropriate construction, if an exact existential specification of a program is given, it is possible to conclude from it an exact universal specification. This is especially beneficial for proof rules that exploits existential specification in some intermittent stages to achieve compositionality. Another motivation is that often the property or the program specified is best presented or proved using existential specification, but universal properties of the program ought to be provable from it.

In Section 2, some preliminary definitions concerning modeling and asserting about programs are reviewed. Then, existential satisfaction is formally presented. In Section 3 some examples of using existential specification for compositional verification are given. In Section 4, exact existential specification of a program is defined and shown to be sensitive to the choice of representing the states. Section 5 provides conditions for semantical constructions that allow exact existential specifications of programs. A construction of such structures is also presented. In Section 6, proof rules are given for generalising from an existential specification into a universal specification. It is shown that using these rules, an exact existential specification can be generalised into an exact universal specification.

2 Universal and Existential Satisfaction

2.1 Interleaving Semantics and Linear Temporal Logic

A program P is a finite set of named operations T, a finite set of variables V, and an initial condition Θ (a first order predicate). Each operation τ is a triple $\langle l_\tau, en_\tau, \mathsf{f}_\tau\rangle$ such that $l_\tau \subseteq V$ is the set of τ's *local* variables, i.e., the only variables that τ can examine and change, en_τ is its enabling condition, i.e., a first order predicate with free variables from l_τ which determines when τ can be executed, and f_τ is a transformation on τ's local variables (which is naturally extended to transform the entire set of program variables V by not changing the variables which are not in l_τ). An operation τ will be denoted by $en_\tau \longrightarrow l_\tau := \mathsf{f}_\tau(l_\tau)$. The assignment will not mention explicitly variables from l_τ that are not changed, e.g., writing $x = y \longrightarrow x := x + 1$ instead of $x = y \longrightarrow (x, y) := (x + 1, y)$.

A state is an assignment (interpretation) of values to the variables. In program verification, the states are usually interpretations of the set of program variables. However, by adding auxiliary variables [7] to the program one can add additional information to each state such as the sequence of operations executed so far. An *interleaving sequence* is a finite or infinite sequence of states. A program P generates a set of interleaving sequences $\xi = \xi_0\,\xi_1\,\xi_2\ldots$ such that ξ_0 satisfies Θ, and for each $0 < i <|\xi|$ ($|\xi| = \omega$ if the sequence is infinite), there exists some operation $\alpha_i \in T$ such that ξ_{i-1} satisfies en_{α_i}, and $\xi_i = \mathsf{f}_{\alpha_i}(\xi_{i-1})$. This sequence is finite iff its last state ξ_n does not satisfy en_τ for any $\tau \in T$. The sequence of operations $\alpha_1\,\alpha_2\ldots$ corresponding to ξ is called the *generating sequence* of ξ.

Let $I \subseteq T \times T$ be a symmetric and irreflexive relation called the *independence* relation, satisfying that $(\alpha, \beta) \in I$ iff $l_\alpha \cap l_\beta = \phi$. (Notice that $l_\alpha \cap l_\beta = \phi$ is usually a necessary condition for α and β to be independent but not always a sufficient condition. However, one can always force any two such operations to be dependent by adding to both sets l_α and l_β a fresh common dummy variable.) If $(\tau_1, \tau_2) \in I$, we say that τ_1 and τ_2 are *independent*, otherwise they are *dependent*. The independence relation I identifies when it is possible to commute operations.

Requirements from execution sequences called 'justice' and 'fairness' [22] are often imposed, so that only those interleaving sequences that satisfy them are considered as representing the executions of a program. We choose the following constraint:

Definition 2.1 *The* execution sequences *of a program P are exactly the interleaving sequences whose states are interpretations of P's variables and satisfy the following* justice *property: if from some state in the sequence, an operation α is enabled, then an operation that is dependent on α (possibly α itself) will occur eventually in the sequence. The set of execution sequences of P is denoted $\widetilde{\mathcal{E}}[\![P]\!]$.*

Linear temporal logic [28] is a formalism that can be used to specify concurrent programs. As usual in LTL, $\Box\varphi$ means that φ holds in every future state in a sequence, $\Diamond\varphi$ means that φ will hold in some future state, $\varphi\mathcal{U}\psi$ means that φ will continue to hold until ψ holds, $\varphi\,\mathsf{U}\psi$ means that if either φ holds in every future state or until ψ holds, and $\bigcirc\varphi$ means that φ holds starting with the next state. Past temporal modals that correspond to the above future modals also exist. For example, $\varphi\mathcal{S}\psi$ (which is the past version of $\varphi\mathcal{U}\psi$) means that φ holds continuously since a state in the past in which ψ held. The connectives '$\neg$', '$\vee$', '$\wedge$', '$\rightarrow$', '$\leftrightarrow$' and the (rigid, i.e., state-independent) quantifiers '$\forall$' and '$\exists$' are also used. The predicate T holds always, while F never holds for every state or sequence.

In [4], a binary modal operator $\mathcal{C}$ called *chop* was added to LTL. The formula $\varphi\mathcal{C}\psi$ is satisfied by an interleaving sequence if it can be partitioned into a prefix satisfying φ and a suffix satisfying ψ or if the entire sequence satisfies φ. An LTL formula φ is interpret over a pair of an interleaving sequence ξ and an index i that designate the starting state in the sequence from which the formula needs to be satisfied. This is denoted $(\xi, i) \models \varphi$. An interleaving sequence ξ satisfies an LTL formula φ iff $(\xi, 0) \models \varphi$. This is denoted also as $\xi \models \varphi$. An LTL formula is *valid* if it holds for each sequence of states. A *past temporal formula* ψ is a formula that does not contain future modals. It is said to be *end-satisfied [6]* by a finite interleaving sequence if it holds in its last state, i.e., $(\xi, |\xi| - 1) \models \psi$.

2.2 Trace Semantics

Partial order semantics can be represented in various ways. One of the most simple representations is Mazurkiewicz's trace semantics [21].

A *history* of a program P is a pair $h = \langle J_0, v\rangle$ where J_0 is a state called the *initial interpretation of* h, and $v \in T^*$ satisfies that there exists a finite prefix of an execution sequence $\xi = J_0 J_1 \ldots J_n$ such that $v = \alpha_0\alpha_1 \ldots \alpha_n$ is its generating sequence. For each history $h = \langle J, v\rangle$, let the n^{th} state J_n in the above sequence (which is a function of J_0 and v) be denoted by fin_h. This is called the *final interpretation* of h.

Two histories $h = \langle J, v\rangle$ and $h' = \langle J, w\rangle$ are *equivalent* (denoted $h \equiv h'$) if it is possible to obtain w from v by repeatedly commuting adjacent independent operations. That is, if there exists a sequence of histories $\langle J, v_1\rangle, \langle J, v_2\rangle, \ldots, \langle J, v_n\rangle$ with $v_1 = v$ and $v_n = w$, and for each $1 \leq i < n$ there exist $u, \bar{u} \in T^*$, $(\alpha, \beta) \in I$ such that $v_i = u\alpha\beta\bar{u}$, $v_{i+1} = u\beta\alpha\bar{u}$. For example, if $T = \{\alpha, \beta, \gamma\}$ and $I = \{(\alpha, \beta), (\beta, \alpha), (\gamma, \beta), (\beta, \gamma)\}$, then $\langle J, \alpha\alpha\beta\beta\gamma\gamma\rangle \equiv \langle J, \alpha\beta\alpha\gamma\beta\gamma\rangle \equiv \langle J, \beta\alpha\alpha\gamma\gamma\beta\rangle$, but $\langle J, \alpha\alpha\beta\gamma\gamma\rangle \not\equiv \langle J, \gamma\gamma\beta\alpha\alpha\rangle$. A *trace* is an equivalence class of histories, denoted by $[J, w]$, where J is the common initial interpretation and $\langle J, w\rangle$ is a member of the equivalence class.

Obviously, if $h = \langle J, u\alpha\beta\bar{u}\rangle$ is a history of P, and $(\alpha, \beta) \in I$, then $h' = \langle J, u\beta\alpha\bar{u}\rangle$ is also a history of P. Moreover, $fin_h = fin_{h'}$. Therefore, the final interpretations of all the histories in a trace are identical. This permits defining fin_σ as fin_h for any $h \in \sigma$. Concatenation between two traces $\sigma_1 = [J_1, v]$ and $\sigma_2 = [J_2, w]$, denoted $\sigma_1\sigma_2$, is defined when $fin_{\sigma_1} = J_2$, and is given by $[J_1, vw]$. The relation '$\sqsubseteq$' between traces is defined as $\sigma_1 \sqsubseteq \sigma_2$ iff there exists some σ_3 such that $\sigma_1\sigma_3 = \sigma_2$. It is then said that σ_1 is *subsumed* by σ_2 or that σ_1 is a *prefix* of σ_2. If, in addition, σ_3 contains a single operation, σ_2 is an *immediate successor* of σ_1.

An *accessible* trace of a program P is any trace obtained as the equivalence class of some of the histories of P with an initial interpretation satisfying Θ. The traces *generated by a program* P are traces σ such that for some ρ, $\rho\sigma$ is an accessible trace of P. In the sequel we consider only traces generated by programs.

Two traces σ_1, σ_2 are *consistent*, denoted $\sigma_1 \Uparrow \sigma_2$, iff there exists σ_3 such that $\sigma_1 \sqsubseteq \sigma_3$ and $\sigma_2 \sqsubseteq \sigma_3$. A *run* (partial order execution) Π of P is a maximal set of pairwise consistent accessible traces. The set of all runs of a program P is denoted R_P. A set of traces Π is *directed* if for each two traces σ, $\rho \in \Pi$, a trace subsuming both traces exists in Π. It is not difficult to show that each run Π is also prefix closed and maximal consistent.

For each run Π, an *interleaving sequence of traces* is a sequence whose states are traces $\sigma_0\, \sigma_1\, \sigma_2 \ldots$ of Π, such that $\sigma_0 = [J, \epsilon]$ (an empty trace) for some $J \models \Theta$, and for each $i \geq 0$, $\sigma_i \sqsubset_{im} \sigma_{i+1}$.

Definition 2.2 *An* observation *of* Π *is an interleaving sequence where for each* $\sigma \in \Pi$, *there exists some* σ_i, $i \geq 0$ *such that* $\sigma \sqsubseteq \sigma_i$.

The set of observations of Π is denoted $obs(\Pi)$. Denote the fact that o, o' are both in $obs(\Pi)$ for some $\Pi \in R_P$ by $o \approx o'$. This is obviously an equivalence relation on the observations of each program P. Denote $\Delta_P = \bigcup_{\Pi \in R_P} \Pi$, i.e., the set of all the traces of the program P. If Σ is a set of traces, denote $\downarrow\Sigma = \{\sigma \mid \exists\rho \in \Sigma(\sigma \sqsubseteq \rho)\}$ (i.e., all the traces subsumed by traces of Σ).

It is shown in [17, 25] that the set of observations of a program are exactly the interleaving sequences of traces satisfying the justice condition of Definition 2.1. Another important characterisation of the observations of P is given by the following Lemma.

Lemma 2.3 *An interleaving sequence of traces $\xi = \sigma_0 \sigma_1 \sigma_2 \ldots$ of P is an observation iff for each trace $\sigma \in \Delta_P$, there exists some trace σ_i on ξ, such that either $\sigma \sqsubseteq \sigma_i$ or $\sigma_i \not\uparrow \sigma$. In particular, this holds for $\Sigma = \Delta_P$. Moreover, this still holds if Δ_P is replaced by any set of traces Σ such that $\downarrow \Sigma = \Delta_P$.*

Proof. Follows from the definition of runs and observations. ∎

In order to reason about sequences of traces using temporal logic, it is necessary to map traces into states, such that state predicates and functions symbols on state variables can be used to assert about traces, and temporal operators can then be added to formalise sequence assertions. These state predicates and functions can refer to information extracted from traces. For example, in specification and verification of concurrent programs, often only the value of the program variables according to fin_σ (i.e., the values of the variables after executing operations according to any history in σ) are of interest.

Definition 2.4 *A* representation of traces *is a mapping $f : \Gamma \mapsto \mathcal{R}$ from a class of traces Γ to a set of elements (the domain of the representation) $\mathcal{R}$. The function f is naturally generalised to sequences and sets of sequences. Sequences of $\mathcal{R}$-elements will be called $\mathcal{R}$-sequences.*

For trace semantics, $\mathcal{E}[\![P]\!] = \{f(obs(\Pi)) \mid \Pi \in R_P\}$ represents not only the set of observations of P, but also the grouping of them into sets. Hence, if f maps traces into interpretations of program variables, then $\widetilde{\mathcal{E}}[\![P]\!] = \bigcup_{\Pi \in R_P} \mathcal{E}[\![P]\!]$. That is, $\widetilde{\mathcal{E}}[\![P]\!]$ is a single set containing all the sequences of $\mathcal{E}[\![P]\!]$ (but not the partitioning). Denote by $\xi \Subset \mathcal{E}[\![P]\!]$ the fact that $\xi \in \bigcup_{\Pi \in R_P} \mathcal{E}[\![P]\!]$.

2.3 Universal and Existential Satisfaction

Definition 2.5 *An LTL formula φ is* universally satisfied *by a set of sequences M, denoted by $M \models \varphi$, iff for each sequence $\xi \in M$, $\xi \models \varphi$. This is extended to a set of sets of sequences $\mathcal{A}$, denoting $\mathcal{A} \models \varphi$ iff $M \models \varphi$ for each $M \in \mathcal{A}$. (Notice that the meaning of the relation '$\models$' depends on the type of its first argument.)*

Definition 2.6 *An LTL formula φ is* existentially satisfied *by a set of sequences M, denoted by $M \overset{\exists}{\models} \varphi$, iff there exists some sequence $\xi \in M$ such that $\xi \models \varphi$. This is extended to a set of sets of sequences $\mathcal{A}$, denoting $\mathcal{A} \overset{\exists}{\models} \varphi$ iff $M \overset{\exists}{\models} \varphi$ for each $M \in \mathcal{A}$.*

Thus, $\mathcal{E}[\![P]\!] \overset{\exists}{\models} \varphi$ means that φ is satisfied by at least one representative observation from any equivalence class of $\mathcal{E}[\![P]\!]$, while $\mathcal{E}[\![P]\!] \models \varphi$ ignores the equivalence relation by requiring that a formula is satisfied by all the (representations of the) observations of P (i.e., all the sequences $\xi \Subset \mathcal{E}[\![P]\!]$ satisfy the formula φ). Existential satisfaction is weaker than universal satisfaction with respect to the same formula, because it demands that only *representatives* of the equivalence classes satisfy a formula, rather than all of them. Denote by $P \vdash \mu$ the fact that one can prove using some proof system that $\mathcal{E}[\![P]\!] \models \mu$. Similarly, $P \overset{\exists}{\vdash} \mu$ means that $\mathcal{E}[\![P]\!] \overset{\exists}{\models} \mu$ is provable.

The same LTL formula can be interpreted in both universal and existential ways. Henceforth, we will mention explicitly the way a formula is interpreted over the semantic models. The expressive power [9, 18] of existential and universal satisfaction is incomparable. This follows from the fact that if N and M are two sets of sequences such that $N \subseteq M$, then there exists no formula which is satisfied existentially by N but not by M (even in the cases where there exists such a universally satisfied formula), and no formula which is satisfied universally by M but not by N (even in the cases where there exists such an existentially satisfied formula).

3 Applicability of Existential Specification

Existential specification is more appropriate than universal specification when some execution sequences of a program correspond better than others to the property under consideration. In these cases, it is preferable to specify the behaviour of at least one sequence out of each equivalence class.

Composing a program out of smaller segments often results in a program whose behaviour is best described existentially. The reason for this is that by composing, concurrency is enhanced and the executions of the segments are interleaved. However, some of the concurrent activities of different parts are independent of each other. Thus, sequences in which a pair of activities are considered to execute separately one after the other in one of either orders are equivalent to all the sequences in which occurrences of operations from both activities are interleaved (and thus can be chosen as representatives to this larger set of sequences).

In order to achieve compositionality, we use *program segments* rather than programs. A program segment differs from a program by not having its own initial condition. The pair $\langle P, \Theta_P \rangle$ of a program segment and a state predicate Θ_P is treated similarly to a program with an initial condition Θ_P, except that Θ_P is not restricted to be satisfied by representations of empty traces. Thus, the execution of such a pair can be modeled by runs that can start with a non-empty trace.

A trivial example is the composition of two terminating processes (segments) P and Q which do not interact, i.e., all the operations from P are independent of all the operations of Q. Then, $\mathcal{E}[\![\langle P \mid\mid Q, \Theta_{PQ} \rangle]\!]$ includes all the interleavings of the execution sequences from P and from Q. However, each execution sequence of $\langle P \mid\mid Q, \Theta_{PQ} \rangle$ is equivalent to an execution sequence that consists of some (finite) execution sequence of $\mathcal{E}[\![\langle P, \Theta_{PQ} \rangle]\!]$ followed by an execution sequence of $\mathcal{E}[\![\langle Q, \Theta_Q \rangle]\!]$. If $\langle P, \Theta_{PQ} \rangle \vdash^{\exists} \varphi \wedge \Diamond(\theta_P \wedge \bigcirc \mathrm{F})$ (i.e., φ holds when P is executed, and P terminates with a state satisfying the state predicate θ_P) and $\langle Q, \Theta_Q \rangle \vdash^{\exists} \psi$, and $\theta_P \rightarrow \Theta_Q$ (i.e., the condition Θ_Q allows executing Q from any state in which the execution of the segment P can terminate), then $\langle P \mid\mid Q, \Theta_{PQ} \rangle \vdash^{\exists} \varphi \mathcal{C} \psi$. This can be formalised as a proof rule:

$$\frac{\begin{array}{l} \langle P, \Theta_{PQ} \rangle \vdash^{\exists} \varphi \wedge \Diamond(\theta_P \wedge \bigcirc \mathrm{F}) \\ \langle Q, \Theta_Q \rangle \vdash^{\exists} \psi \\ \theta_P \rightarrow \Theta_Q \end{array}}{\langle P \mid\mid Q, \Theta_{PQ} \rangle \vdash^{\exists} \varphi \mathcal{C} \psi} \tag{1}$$

The existential '$\vdash^{\exists}$' in the antecedents of this proof rule can be replaced with a universal '$\vdash$', but not in the consequence.

Although the above example may be considered simple and untypical, the following are examples of general cases to which similar verification techniques can be applied. First, consider the composition of CSP processes in the partial correctness proof rules of [2]. There, it is observed that composing the segment $S_1; \alpha; S_2$ in one process with $S_3; \bar{\alpha}; S_4$ in the second, where α and $\bar{\alpha}$ are matching send and receive communication commands and $S_1 \ldots S_4$ are local segments, behaves as $S_1; S_3; \alpha \mid\mid \bar{\alpha}; S_2; S_4$. The soundness of the proof rules in [2] relies on the fact the set of executions in which the segments are executed in this order contains enough representatives equivalent to all other executions, and that if at least one sequence from each equivalence class satisfies the partial correctness property, then all the other sequences also satisfy it.

Another way to compose programs is sequentially as *communication closed layers* [8]. There, two layers $S = [S_1 \mid\mid \ldots \mid\mid S_n]$ and $M = [M_1 \mid\mid \ldots \mid\mid M_n]$ are composed into a program $S; M = [S_1; M_1 \mid\mid \ldots \mid\mid S_n; M_n]$. If there is no possible communication between any S_i and M_j for $1 \leq i, j \leq n$, the following compositional proof rule can be formulated:

$$\frac{\begin{array}{l}\langle S,\, \Theta_S\rangle \models^{\exists} \eta \wedge \Diamond(\theta_S \wedge \bigcirc \mathbf{F}) \\ \langle M,\, \Theta_M\rangle \models^{\exists} \delta \\ \theta_S \rightarrow \Theta_M\end{array}}{\langle S;\, M,\, \Theta_S\rangle \models^{\exists} \eta \mathcal{C} \delta} \tag{2}$$

This rule reflects the fact that for each equivalence class of $\langle S;\, M,\, \Theta_S\rangle$, there exists a representative sequence in which M is executed entirely after S [15]. It is interesting to observe that although the layered composition and the concurrent composition are rather different, the structure of their proof rules is identical. Here as before, the existential '$\models^{\exists}$' in the antecedents of the proof rule can be replaced with a universal '$\vdash$', but not in the consequence. This simple compositional proof rule (compare [4]) allows one to compose provably correct programs from layers. This rule does not restrict the formulas η and δ (but it requires that S will terminate and that the layers are compatible to execute one after the other). This extends proof rules for layered programs that deal only with partial correctness [8] and total correctness [16]. Moreover, using the results of Section 6, compositional completeness [31] of the rules (1) and (2) can be shown, i.e., that every universal property of the program can be deduced from properties of S and M using this rule and pure temporal logic reasoning.

The last example is to consider transforming a program which was designed to operate in a fault-free environment into a fault-tolerant program that is able to recover from some expected faults [32]. A transformation exists that adds recovery handlers to each such basic program. These handlers are based upon taking snapshots [5] of the global state of the program from time to time and retracting upon the occurrence of a failure to the last snapshot taken. For such a program transformation, a *specification transformation* can be formalised, converting properties of the basic program into properties of its fault-tolerant version. Thus, verification of the fault-tolerant version can be done compositionally, by proving properties of the basic program, applying the specification transformation and doing some pure temporal verification. There is no need to actually verify that the property is satisfied by using the code of the fault-tolerant version of the program directly. Moreover, the verification of the specification transformation is done only once for all possible programs [24]. The snapshot taken in some execution sequence does not necessarily correspond to a global state that occurred in the past of the same sequence. However, there always exists an equivalent sequence in which this is true. Thus, the outcome of the formula transformation can be conveniently given by an existentially satisfied formula.

4 Exact Universal and Existential Specification of Programs

4.1 Exact Specification of a Program

Our aim is to determine the ability of a specification formalism to describe the structures (be it execution sequences or equivalence classes of them) modeling the behaviour of each given program P. This means that for any program P, there exists a formula φ such that

- all the structures modeling P satisfy φ, and
- all the structures that satisfy φ represent executions of P.

We call such a formula φ an *exact specification* of P (with respect to the class of structures used to model P) [19]. An exact specification φ of a program P in some formalism $\mathcal{L}$ has the property that every formula $\psi \in \mathcal{L}$ that holds for P is a consequent of φ, i.e., each structure that satisfies φ satisfies also ψ. This is denoted $\varphi \models \psi$.

This is especially important for achieving completeness of compositional verification methods [4, 31], i.e., proofs in which segments of a program are verified separately and then the proofs are combined (instead of proving each property with respect to the entire program). Completeness proofs of compositional methods often use the following conditions:

1. It is known that for each pair of programs (or program segments) P_1 and P_2 there exist exact specifications, say, φ_1 and φ_2, respectively.
2. It is possible to verify from φ_1 and φ_2 using some compositional proof rule [4] that ψ holds for the program P that is combined from P_1 and P_2.
3. ψ is an exact specification of P (i.e., the composition of the specifications preserves exactness).

Given the above conditions, the compositional proof method under discussion is compositional complete relative to verification in $\mathcal{L}$: since every property η of the combined program P is a consequence of ψ, it can be proved that P satisfies η by properties of P_1 and P_2 as follows:

(i) $P \vdash \psi$ (with ψ an exact specification of P) is proved using the compositional proof rule from exact specifications of P_1 and P_2.
(ii) $\psi \rightarrow \eta$ is proved in $\mathcal{L}$.

Then, $P \vdash \eta$ follows by a simple deduction rule from (i) and (ii).

For universal specification we define exact specification over sets of sequences by ignoring the partitioning into sets.

Definition 4.1 *A formula Φ_P is an* exact universal specification of a program *P if for each interleaving sequence ξ, $\xi \models \Phi_P$ iff $\xi \Subset \mathcal{E}[\![P]\!]$.*

It was shown in [19] that for interleaving semantics, for each program P, there exists a temporal formula that is satisfied exactly by its set of executions (i.e., universal satisfaction).

4.2 Exact Existential Specification

For existential specification, similarly to exact universal specification, for each program P, there should exist a formula that is satisfied exactly by equivalence classes of sequences of P. However, this is not as straightforward to achieve as in the universal case. Consider the following definition:

Definition 4.2 (first attempt) *A formula Υ_P is an* exact existential specification *of a program P iff for each set of sequences M, $M \overset{\exists}{\models} \Upsilon_P$ iff $M \in \mathcal{E}[\![P]\!]$.*

According to this definition, exact existential specification does not exist for any program. This is because one can add arbitrary new sequences to any set of sequences M such that $M \overset{\exists}{\models} \Upsilon_P$, obtaining $M' \notin \mathcal{E}[\![P]\!]$ such that $M' \overset{\exists}{\models} \Upsilon_P$ (by Definition 2.6, since $M' \subseteq M$). Thus, there is a need to limit the sets of sequences under consideration.

Definition 4.3 (second attempt) *A formula Υ_P is an* exact existential specification of a program *P iff for each set of sequences M obtained as an equivalence class of sequences of* some *program, $M \overset{\exists}{\models} \Upsilon_P$ iff $M \in \mathcal{E}[\![P]\!]$.*

Again, it can be shown that it is not true that under all representations for each program there exists an exact existential specification.

Example. Consider the most obvious representation, where each trace σ is represented by the values of the program variables in fin_σ. We will show that there is a program P_1 for which there is no exact existential specification. Consider the following two programs P_1 and P_2: The program P_1 has two interdependent operations $\alpha : x = y \longrightarrow x := x + 1$, and $\beta : x > y \longrightarrow y := y + 1$. The program P_2 has two independent operations $\alpha : true \longrightarrow x := x + 1$, and $\beta : true \longrightarrow y := y + 1$. Both programs are initiated with $x = 0$ and $y = 0$.

The programs have a mutual interleaving sequence, namely $(x = 0,\ y = 0) \xrightarrow{\alpha} (x = 1,\ y = 0) \xrightarrow{\beta} (x = 1,\ y = 1) \xrightarrow{\alpha} (x = 2,\ y = 1) \xrightarrow{\beta} \ldots$, denoted according to its generating sequence as $(\alpha\beta)^\omega$, where u^ω means u occurring infinitely many times. That is, x and y are incremented in turn, one after the other. However, for P_1, there is only one interleaving sequence, while for P_2, the operations can be selected to repeat arbitrarily and all the sequences $(\beta^*\alpha\alpha^*\beta)^\omega$ are equivalent. Hence, any formula such as

$$\begin{aligned}&(\Box 0 \leq x - y \leq 1) \wedge \forall z\,(\Diamond x > z) \wedge \\ &\quad \Box \forall z\, \forall t\,(\,(x = z \wedge y = t) \rightarrow \bigcirc(\,(x = z + 1 \wedge y = t) \vee (x = z \wedge y = t + 1)\,)\,),\end{aligned}$$

that is satisfied exactly by the the single interleaving sequence of P_1, is not an exact existential specification of P_1. This follows from the fact that P_2's executions constitute a single equivalence class that properly contains P_1's single interleaving sequence, and thus also existentially satisfies any such formula. ▟

Another problem of the attempted Definition 4.3 is that the phrase "set of sequences M obtained as the equivalence class of sequences of *some* program" refers to either adding some program-dependent information to the sequences, or the ability of showing that a set of sequences is obtainable as an equivalence class of some program (by axiomatising the properties of such classes). Both of these are undesirable: adding additional information about the program defies the very idea of giving the exact specification of a program by a formula; alternatively, proving that an equivalence class corresponds to some program would require additional effort.

Instead, using some appropriate representation, it should be possible to define a condition on sets of representations sequences that limits their scope. Such a condition will later help to guarantee that if one of the sequences in a set is obtained as the representation of an execution sequence of some program, the rest of the sequences are exactly all the representations of sequences equivalent to it.

Definition 4.4 *Let B be a condition (expressed in some predefined formalism) on sets of $\mathcal{R}$-sequences. Such a condition is called a* bounding condition. *Each set of $\mathcal{R}$-sequences satisfying B is called a B-set.*

A bounding condition B can be seen as a program-independent restriction, limiting the scope of sets of possible $\mathcal{R}$-sequences. (It may allow B-sets that contain sequences that do not represent execution sequences of programs.) It is now possible to redefine exact existential specification using a bounding condition.

Definition 4.5 *A formula Υ_P is an* exact existential specification of a program P with respect to a bounding condition B *if for each B-set M, $M \overset{\exists}{\models} \Upsilon_P$ iff $M \in \mathcal{E}[\![P]\!]$.*

5 Obtaining Exact Existential Specification

In this section, we define some general conditions which guarantee the existence of a bounding condition that facilitates exact existential specification. Then we construct a representation of traces that satisfies these conditions. The following goals are considered:

- No additional information about the modeled program is needed other than the existential specification (e.g., the independence relation).
- The construction should be obtainable using a syntactical transformation of the program, and then applying a simple familiar semantical construction (such as in [21] or [17]). Such a transformation should be as simple as adding to the program a set of appropriate auxiliary variables [7]. This will allow existing verification methods to be adopted.
- Generalisation, i.e., inferring a universal specification from an existential specification, must be possible. This is useful for the cases where an exact specification of a program (e.g., a result of superimposing programs) is best given in existential form, but some interesting properties of the superimposed program are of universal form.

5.1 Requirements from the Representation

We consider now some requirements that are imposed on any domain $\mathcal{R}$ of elements that represent a class of traces.

R1 The relations '$\sqsubseteq$' and '$\sqsubset_{im}$' between traces must be isomorphically definable on the representations. That is, for each pair of traces σ_1, σ_2 of a program P, $\sigma_1 \sqsubseteq \sigma_2 \Leftrightarrow subsumed(f(\sigma_1), f(\sigma_2))$, where $subsumed(s, t)$ is the corresponding relation defined among representations. Similarly, $\sigma_1 \sqsubset_{im} \sigma_2 \Leftrightarrow im\_subsumed(f(\sigma_1), f(\sigma_2))$.

Thus, a representation function f that identifies all the traces with the same final interpretation by assigning them to the same $\mathcal{R}$-element is inappropriate, as a triple $\sigma_1 \sqsubseteq \sigma_2 \sqsubseteq \sigma_3$, $\sigma_1 \neq \sigma_3$, with σ_1 and σ_3 having the same representation must not exist.

Since no additional information about the program need to be given together with an existential specification we require:

R2 The predicates $subsumed$ and $im\_subsumed$, defined over the representation elements, must be program-independent. That is, if $subsumed(s, t)$, then for *every* program P with traces σ and ρ such that $f(\sigma) = s$ and $f(\rho) = t$ it must be that $\sigma \sqsubseteq \rho$. (Notice that '$\sqsubseteq$' *is* program-dependent.) Moreover, when $subsumed(s, t)$ holds for some $t = f(\sigma)$, where $\sigma \in \Delta_P$, then $s = f(\rho)$ for some $\rho \sqsubseteq \sigma$. The same must hold for the relation '$\sqsubset_{im}$' and the corresponding relation $im\_subsumed(s, t)$ among $\mathcal{R}$-elements.

If X is a set of $\mathcal{R}$-elements, denote $\downarrow X = \{s \mid \exists t \in X \wedge subsumed(s, t)\}$. Observe that by **R2**, for any set of traces Σ, $f(\downarrow\Sigma) = \downarrow f(\Sigma)$. Denote by $\langle\!\langle\xi\rangle\!\rangle$ the set of elements that appear on the sequence ξ.

In the sequel, we will fix B as the following condition on sets of $\mathcal{R}$-sequences M: let ξ be some $\mathcal{R}$-sequence of M. Then, B is the condition that M includes *exactly all* the sequences of the form $s_0\, s_1\, s_2 \ldots$ that satisfy the following conditions:

B1 $\neg\exists s\,(subsumed(s, s_0) \wedge (s \neq s_0))$ *(i.e., s_0 represents an empty trace),*

B2 *for each* $0 \leq i < |\xi'|$, $s_i \in \downarrow \langle\!\langle\xi\rangle\!\rangle$,

B3 *for each* $0 < i < |\xi'|$, $im\_subsumed(s_{i-1}, s_i)$, *and*

B4 *for each* $s \in \langle\!\langle\xi\rangle\!\rangle$, *there exists some* $0 \leq i < |\xi'|$ *such that* $subsumed(s, s_i)$.

Notice that B already contains an appropriate restriction on adjacent elements.

Lemma 5.1 *If M is a B-set that contains an $\mathcal{R}$-sequence $\xi = f(o)$ for some observation o of P, then M is exactly the set of $\mathcal{R}$-sequences $\{\xi' \mid \xi' = f(o') \wedge o \approx o'\}$ (i.e., M is the set of $\mathcal{R}$-sequences representing all the observations of the same run as o).*

Proof. Let ξ be an $\mathcal{R}$-sequence representing some observation o of some run $\Pi \in R_P$. Then, it is easy to check that by **R1** and **R2**, $f(\downarrow \langle\langle o \rangle\rangle) = \downarrow \langle\langle \xi \rangle\rangle = f(\Pi)$. Thus by B1–B4 and using the definition of observations 2.2, the sequences in M are exactly the $\mathcal{R}$-sequences representing the observations of Π (i.e., the observations that are equivalent to o). ▟

For convenience, we fix some variable c to refer to the 'current state' when appearing in a temporal formula. The following notation will be used in the sequel:

- $consistent(s, t)$ – holds iff $\exists r\,(subsumed(s, r) \wedge subsumed(t, r))$. Notice that if $\sigma_1 \Uparrow \sigma_2$ for *some* program P, then $consistent(f(\sigma_1), f(\sigma_2))$.
- $[\![\varphi]\!]$ – the set of elements satisfying φ.
- $Subsumed_\varphi$ – a predicate transformer returning a predicate that is satisfied by exactly the elements subsumed by elements satisfying φ, i.e., $[\![Subsumed_\varphi]\!] = \downarrow [\![\varphi]\!]$. Both φ and $Subsumed_\varphi$ are state predicates (i.e., do not contain temporal modals). $Subsumed_\varphi$ can be defined as $\exists t\,(subsumed(s, t) \wedge \varphi(t))$ (where s is its free variables that stands for an $\mathcal{R}$-element, i.e., $Subsumed_\varphi(s)$ is a predicate of s).
- $Maximal$ – The temporal formula

$$\Box \forall s\,(im\_subsumed(\mathrm{c}, s) \rightarrow \Diamond(subsumed(s, \mathrm{c}) \vee \neg consistent(s, \mathrm{c}))).$$

The requirements **R1** and **R2** assures that the relations $\sqsubseteq$, $\sqsubset_{im}$ between traces are correspondingly definable among their representations. The third requirement allows defining maximal sets of $\mathcal{R}$-elements that correspond to the runs of programs. It is not true that this is already guaranteed by the requirements **R1** and **R2**: although the relation *subsumed* between representations corresponds to '$\sqsubseteq$' (and the maximality of the runs is based upon the relation '$\sqsubseteq$'), the representations of the traces Δ_P of a single run are embedded in $\mathcal{R}$ with representations of other runs.

R3 If $\Pi \in R_P$ for some program P, then the set of $\mathcal{R}$-elements $f(\Pi)$ satisfies that any representation which immediately subsumes an element that belongs to $f(\Pi)$, is either itself in $f(\Pi)$ or is inconsistent with some element of $f(\Pi)$. This is formally written as:

$$\forall t\,((t \notin f(\Pi) \wedge \exists r\,(im\_subsumed(r, t) \wedge r \in f(\Pi))) \rightarrow \exists s \in f(\Pi)\, \neg consistent(t, s)).$$

Lemma 5.2 *A sequence of elements $\xi = s_0 s_1 \ldots$ from $f(\Delta_P)$ such that $\forall i, 0 \leq i < |\xi|, subsumed(s_i, s_{i+1})$ is a representation of an observation of P (i.e., $\xi = f(o)$ for some $o \in obs(\Pi), \Pi \in R_P$) iff $\xi \models Maximal$.*

Proof. Assume first that $\xi = f(o)$ for some observation o of Π. Let $t \in \langle\langle \xi \rangle\rangle$ be an element satisfying $im\_subsumed(t, s)$. If $s = f(\sigma)$ for $\sigma \in \Pi$ then by Definition 2.2, there exists a trace $\rho \in \langle\langle o \rangle\rangle$ that subsumes σ, and thus $f(\sigma) \in \langle\langle \xi \rangle\rangle$, and $subsumed(s, f(\rho))$. Otherwise, by **R3** there exists some element $\rho \in \Pi$ such that $\neg consistent(t, f(\rho))$. Since o is an observation, there exists a trace $\rho' \in \langle\langle o \rangle\rangle$ that subsumes ρ. It is easy to see that since $\neg consistent(t, f(\rho)) \wedge subsumed(f(\rho), f(\rho'))$, then $\neg consistent(t, f(\rho'))$.

To prove the other direction, assume that $\xi \models Maximal$. Suppose that some operations α is enabled in P after some trace σ of $\langle\langle o \rangle\rangle$. Then, $\sigma[\alpha] \in \Delta_P$. According to $Maximal$, there exists some element $t = f(\rho)$, $\rho \in \langle\langle \xi \rangle\rangle$ such that either $subsumed(f(\sigma[\alpha]), t)$ or $\neg consistent(f(\sigma[\alpha]), t)$, and thus in the latter case, $\sigma[\alpha] \not\Uparrow \rho$. From properties of traces it can be shown that both cases correspond to the fact that an operation that is dependent on α (including α itself, in the former case) is executed in ξ (and is contained in $f(\rho)$). This condition on interleaving sequences of traces of P was shown in [17, 25] to be equivalent to the definition of observations. Thus, ξ represents an observation. ▟

The following theorem provides a useful form of exact existential specification:

Theorem 5.3 *If $[\![\varphi]\!] \subseteq f(\Delta_P)$, and $\mathcal{E}[\![P]\!] \overset{\exists}{\models} Maximal \wedge \Box\varphi$, then $Maximal \wedge \Box\varphi$ is an exact existential specification of P with respect to B.*

Proof. Let $\eta = Maximal \wedge \Box\varphi$. Let M be an arbitrary B-set such that $M \overset{\exists}{\models} \eta$. Then there exists an $\mathcal{R}$-sequence $\xi \in M$ satisfying η. Since $[\![\varphi]\!] \subseteq f(\Delta_P)$, all the elements in $\langle\!\langle \xi \rangle\!\rangle$ are $\mathcal{R}$-elements representing traces of P. From B3, for each adjacent elements $s_i, s_{i+1} \in \langle\!\langle \xi \rangle\!\rangle$, $im\_subsumed(s_i, s_{i+1})$. Thus, from Lemma 5.2 it follows that ξ is an observation of P. It follows from Lemma 5.1 that all the sequences in M represent observations of P as well. ■

For each program P, a predicate φ_P that includes exactly the elements of $f(\Delta_P)$ is obtainable by a standard construction (it is used as a strongest invariant of a program in standard completeness proofs, e.g., [20]). This predicate satisfies the conditions of Theorem 5.3. Thus, an exact existential specification *always exists*. However, this predicate φ_P is not necessarily the best choice (or the only one) to be used for exact existential specification, as for this choice of φ_P, $Maximal \wedge \Box\varphi_P$ is both an exact existential and an exact universal specification for P.

In addition to the above requirements, it is also required that the representations must contain in a retrievable form any information that is needed for formulating the intended program properties. (In particular, the initial condition of the program must be interpretable over the representations.) For example, if the assertion language specifies properties of a program by asserting on its variables, then the value of these variables must be retrievable from the trace representation such that relation and functions on these values can be defined. Alternatively, one might be interested only in formulating properties of the sequences of operations occurred [13].

5.2 A Construction of a Representation

A construction of representations for traces that satisfy the requirements **R1**, **R2** and **R3** will now be presented. Its main purpose is to demonstrate that these conditions are satisfiable. There is no claim that this is the best or most efficient choice of representation for any other purpose.

It is sometimes convenient to represent a trace by denoting one of its histories. This is followed here, when histories, rather than traces, are actually the objects that are represented directly. Predicates on history representations, such as $subsumed$ and $consistent$, can treat $\mathcal{R}$-elements as traces, by using in their definition an equivalence relation between $\mathcal{R}$-elements which is defined for this purpose. Hence, there can exist multiple representations for each trace and observation. This can be easily avoided by assigning unique weights to the $\mathcal{R}$-elements, and then representing each trace by the minimal $\mathcal{R}$-element corresponding to a history of a given trace.

A *snapshot* is finite set of pairs (v, a), where v is a variable and a is a value. It represents a valuation of a set of variables. An *event* is an occurrence of some operation $\tau \in T$. It is represented as a pair containing an operation name τ and a snapshot that includes a pair (v, a), for each $v \in l_\tau$. For each such pair, a is the value of v just after executing τ. (Notice that the name of the variable can be represented by a string or an integer.)

An $\mathcal{R}$-element is then a triple $\ll J, \underline{E}, U \gg$, where J is a snapshot, E is a finite sequence of events (underlined for convenience of reading), and U is a set of pairs, each one containing the name τ of the operation that is enabled after the occurrence of the events in E and its set of variables l_τ. The following restrictions guarantee that each element in $\mathcal{R}$ represents a history of some program: (1) the set of variables of the snapshot J must include at least all the variables that appear in the events E, and (2) each two events with the same name have snapshots of the same variables. This agreement on the variables must also hold for any event and an element of U with the same name of operation.

As an example, the program P_1 that appear in the example of Section 4 has an $\mathcal{R}$-element

$$\ll \{(x,0),\,(y,0)\},\underline{\langle\alpha,\,\{(x,1),\,(y,0)\}\rangle\,\langle\beta,\,\{(x,1),\,(y,1)\}\rangle\,\langle\alpha,\,\{(x,2),\,(y,1)\}\rangle},\,\{\langle\beta,\{x,y\}\rangle\}\gg, \quad (3)$$

while P_2 has

$$\ll \{(x,\,0),\,(y,\,0)\},\underline{\langle\alpha,\,\{(x,\,1)\}\rangle\,\langle\beta,\,\{(y,\,1)\}\rangle\,\langle\alpha,\,\{(x,\,2)\}\rangle},\{\langle\alpha,\,\{x\}\rangle,\,\langle\beta,\,\{y\}\rangle\}\gg. \quad (4)$$

Both $\mathcal{R}$-elements correspond to the execution of three events: executing α, β and then α again.

The independence between pairs of operations can be easily translated using this representation into a corresponding independence of their events: Two events are independent iff they have no variable in common. For example, the occurrences of α and β in (3) are dependent, as they both have in common the variables x and y, while the occurrences of α and β in (4) are independent.

It is thus possible to define the relations $subsumed$ and $im\_subsumed$ using only the information that is in the $\mathcal{R}$-elements without additional information about the programs they represent. The third component U_s of an $\mathcal{R}$-element $s = f(\sigma)$ is the set of operations enabled immediately after σ. The relation $im\_subsumed$ (and similarly $subsumed$) must reflect this fact by allowing $im\_subsumed(s,\,t)$ only if t is obtained from s (or an element equivalent to s up to repeatedly commuting independent events) by the occurrence of an additional event with the same operation name and the same set of variables as a member of U_s. This use of the third component is essential to guarantee that the requirement **R3** holds.

6 Generalisation of Existential Specification

In this section, we show how under an appropriate representation of traces that satisfy the requirements **R1**, **R2** and **R3** of Section 5 (such as the one given in Section 5.2), it is possible to generalise from an existential specification into a universal specification. The proof system presented in this section also guarantees that one can infer an *exact* universal specification of a program P from an *exact* existential specification of P. This provides a framework that augment compositional proof rules such as (1) and (2) in Section 3 and guarantees compositional completeness.

Lemma 6.1 *If φ is a temporal property, then there exists a past temporal formula η that end-satisfies a finite sequence ξ iff ξ is a prefix of an interleaving sequence that satisfies φ.*

Proof. It is possible to construct for every property φ that is expressible in LTL an equivalent formula of the form $Safe_\varphi \wedge Live_\varphi$, where $Safe_\varphi$ is of the form $\Box\eta$, with η satisfying the above requirements. This is the well known separation into safety and liveness [1]. The proof of this is very similar to [6, Section 4.11], where a related property is proved. This is based on using Gabbay's separation theorem [10]. ▪

We next assume that the first order logic we use for state formulas allows encoding finite sequences [3, 12]. Specifically, we assume that the following functions and predicate can be expressed:

$Pref(\chi,\,i)$ the prefix of length i of χ.
$Len(\chi)$ the length of χ.
$Seq(\chi,\,i,\,s)$ holds when s is the i^{th} element in the sequence χ (and $0 \le i < Len(\chi)$).

Lemma 6.2 *If φ is a temporal property, then there exists a first order formula $States_\varphi(s)$ that holds exactly for the $\mathcal{R}$-elements s that appear in interleaving sequences that satisfy φ.*

Proof. Let η be the formula constructed from φ in Lemma 6.1. Then define a translation $\mathcal{T}(\kappa, \chi)$ that transforms a temporal formula κ into a first order formula that is applied to the encoded sequence χ. The translation can be defined inductively on the structure of κ. For example, if $\kappa = \kappa_1 \mathcal{S} \kappa_2$, then

$$\mathcal{T}(\kappa,\chi) = \exists i\,(0 < i \leq Len(\chi) \wedge \mathcal{T}(\kappa_2, Pref(\chi,i))\,) \wedge \forall j\,(i < j \leq Len(\chi) \rightarrow \mathcal{T}(\kappa_1, Pref(\chi,j))\,)\,)$$

The details of how to translate other past modals are omitted. Then $States_\varphi(s)$ can be defined as

$$\exists n \exists \chi\,(Len(\chi) = n \wedge \mathcal{T}(\eta,\, \chi) \wedge Seq(\chi,\, n-1,\, s)\,).$$

∎

Consider now the following simple deduction rule for existential specification.

$$\frac{\begin{array}{c}\varphi \rightarrow \psi \\ P \models^{\exists} \varphi\end{array}}{P \models^{\exists} \psi} \tag{5}$$

and the following rule for generalisation, where μ is a state formula

$$\frac{P \models^{\exists} \Box \mu}{P \vdash Maximal \wedge \Box Subsumed_\mu} \tag{6}$$

The soundness of (6) is guaranteed by the following lemma:

Lemma 6.3 *If M is a B-set that contains a representation of an observation of some run Π and $M \models^{\exists} \Box \mu$ such that μ is a state formula, then $M \models Maximal \wedge \Box Subsumed_\mu$.*

Proof. From Lemma 5.1, if M contains an observation of some run $\Pi \in R_P$, then all the sequences of M are exactly $f(obs(\Pi)\,)$. Since $M \models^{\exists} \Box \mu$, there exists a sequence ξ in M such that $\xi = f(o)$, where $o \in obs(\Pi)$ and $\xi \models \Box \mu$. Thus, $\langle\langle \xi \rangle\rangle \subseteq [\![\mu]\!]$. Consider now any other sequence ξ' of M. According the conditions of B, $\langle\langle \xi' \rangle\rangle \subseteq [\![Subsumed_\mu]\!]$ and therefore $\xi' \models \Box Subsumed_\mu$. Each sequence in M, is a representation of an observation of Π, and thus from Lemma 5.2, it satisfies $Maximal$. ∎

The above rules can be used to infer a universal property of P from an existential property φ.

1. Prove in LTL that $\varphi \rightarrow \Box \mu$ for some state formula μ.
2. Use the proof rule (5) to infer that $\Box \mu$ is an existential specification of P.
3. Use the rule (6) to infer that $Maximal \wedge \Box Subsumed_\mu$ is a universal specification of P.

Furthermore, if φ is an exact existential specification of P, then an exact universal specification of P can be obtained from it. This is done by choosing μ as $States_\varphi$. By Lemma 6.2, $\varphi \rightarrow \Box States_\varphi$ is valid, and thus can be proved in LTL. The rule (5) can be used to prove that $P \models^{\exists} \Box States_\varphi$. Then (6) can be used to prove that $P \models^{\exists} Maximal \wedge \Box Subsumed_{States_\varphi}$. Since φ is an exact existential specification, $[\![Subsumed_{States_\varphi}]\!] = \downarrow [\![States_\varphi]\!] = f(\Delta_P)$ ($\downarrow [\![States_\varphi]\!] \subseteq f(\Delta_P)$ since each sequence satisfying φ is a representation of an observation of P, and $\downarrow [\![States_\varphi]\!] \supseteq f(\Delta_P)$ since for each run there exists at least one representation of an observation that satisfies φ). Thus, by Lemma 5.2, $Maximal \wedge \Box Subsumed_{States_\varphi}$ is satisfied exactly by all the representations of observations of P.

Recall that, as was shown in Section 4.1, any universal property of P can be deduced from an exact universal specification such as $Maximal \wedge \Box Subsumed_{States_\varphi}$ relative to proving assertions in LTL.

7 Conclusions and Further Research

Representing properties of concurrent programs using existential specification was introduced. It was shown that given an appropriate representation, an exact existential specification Υ_P of a program P can be given such that all the properties of P that are expressed within this formalism are consequences of Υ_P. Moreover, it is possible to transform such an exact existential specification to an exact universal specification efficiently using a proof system.

A representation of trace semantics that allows exact existential specification and generalisation was demonstrated. Alternative constructions, based on a direct representation of partially ordered sets of events can be given under the same requirements. Moreover, when the class of programs dealt with is more constraint, simpler constructions can be made. In particular, the requirement **R1** of Section 5 does not allow a finite representation of finite state programs. It is interesting to check if weakening **R1** is possible or is possible only in some limited cases. The results of this paper are not confined to the temporal logic formalism and can also be adapted to other formalisms such as Büchi automata.

It was shown that existential specification is convenient for compositionality which involves composing concurrent and layered segments. The ability to generalise exact existential specifications is important for achieving completeness of such compositional methods. Thus, existential specification is suggested for compositional program construction, complementing algebraic methods such as [14].

Finally, notice that the generalising proof rule (6) is not optimal in the following sense: it does not guarantee that the strongest universal property ψ that holds for a program that satisfies a given existential property φ can be inferred from φ (however, it does guarantee this when φ is exact). Thus, although useful for achieving compositional completeness of proof rules such as those presented in Section 3, additional proof rules for generalising existential properties are sought.

Acknowledgements

I would like to thank Mathai Joseph and Nissim Francez for their helpful comments.

References

[1] B. Alpern, F.B. Schneider, Defining Liveness, Information Processing Letters 21, 1985, 181–185.

[2] K. R. Apt, N. Francez, W.P. de Roever, A proof system for communicating sequential processes, ACM Transactions on Programming Languages and Systems, Vol 2, 1980, 359–385.

[3] J. W. deBakker, Mathematical Theory of Program Correctness, Prentice–Hall, Englewood Cliffs, N. J, 1980.

[4] H. Barringer, R. Kuiper, A. Pnueli, Now You May Compose Temporal Logic Specification, Proceedings of 16^{th} ACM Symposium on Theory of Computing, 1984, 51–63.

[5] K. M. Chandy, L. Lamport, Distributed Snapshots: determining the global state of distributed systems, ACM Transactions on Computer Systems 3, 1985, 63–75.

[6] E. Chang, Z. Manna, A. Pnueli, The Safety–Progress Classification, Manuscript 1992.

[7] M. Clint, Program proving: Coroutines, Acta Informatica 2, 1973, 50–63.

[8] Tz. Elrad, N. Francez, Decomposition of Distributed Programs into Communication–Closed Layers, Science of Computer Programming 2 (1982), 155–173

[9] E. A. Emerson, J. Y. Halpern, "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic, Journal of the ACM 33 (1986), 151–178.

[10] D. Gabbay, The declarative past and imperative future, in: B. Banieqbal, H. Barringer, A. Pnueli

(editors), Temporal Logic in Specification, LNCS 398, Springer–Verlag, 1987, 407–448.

[11] P. Godefroid, P. Wolper, Using partial orders for the efficient verification of deadlock freedom and safety properties, Proceedings of Computer–Aided Verification, Aalborg, Denmark, 1991.

[12] D. Harel, First order Dynamic Logic, Lecture Notes in Computer Science 68, Springer–Verlag, 1979.

[13] C. A. R. Hoare, Communicating Sequential Processes, Prentice–Hall, 1985.

[14] W. Janssen, J. Zwiers, Protocol Design by Layered Decomposition, A compositional Approach, 2nd Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Nijmegen, The Netherlands, 1992, LNCS 571, Springer–Verlag, 307–326.

[15] S. Katz, D. Peled, Interleaving Set Temporal Logic, Theoretical Computer Science 75(1990),21–43.

[16] S. Katz, D. Peled, Verification of Distributed Programs using Representative Interleaving Sequences, to appear in Distributed Computing.

[17] M. Z. Kwiatkowska, Fairness for Non–interleaving Concurrency, Phd. Thesis, Faculty of Science, University of Leicester, 1989.

[18] "Sometime" is Sometimes "Not Never" – On the Temporal Logic of Programs, Proceedings of the 7th ACM symposium on Principles of Programming Languages, 1980, 174–185.

[19] Z. Manna, A. Pnueli, How to Cook a Temporal Proof System for Your Pet Language. Proceedings of the Symposium on Principles on Programming Languages, Austin, Texas, 1983, 141–151.

[20] Z. Manna, A. Pnueli, Completing the temporal picture, Theoretical Computer Science 83(1991), 97–130.

[21] A. Mazurkiewicz, Traces, Histories, Graphs: Instances of a process monoid, in M. Chytil (Ed.), Mathematical Foundation of Computer Science, LNCS 176, Springer-Verlag, 1984, 115–133.

[22] D. Park, On the Semantics of Fair Parallelism, in D. Biorner (ed.), Proceedings on Abstract Software Specification, LNCS 86, Springer–Verlag, 1979, 504–526.

[23] D. Peled, S. Katz, and A. Pnueli, Specifying and Proving Serializability in Temporal Logic, LICS 91', Amsterdam, The Netherlands, July 1991, 232–245.

[24] D. Peled, M. Joseph, A Compositional Approach to Fault Tolerance Using Specification Transformation, Manuscript, 1992.

[25] D. Peled, A. Pnueli, Proving partial order liveness properties, in M.S. Paterson (ed.), Proceedings of the 17th ICALP, LNCS 443, Springer–Verlag, 1990, 553–571.

[26] W. Penczek, A Concurrent Branching Time Temporal Logic, Conference on Computer Science Logic, LNCS 440, Springer–Verlag, 1989, 337–354.

[27] S. Pinter, P. Wolper, A Temporal Logic for Reasoning about Partially Ordered Computations, Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., August 1984, 23–27.

[28] A. Pnueli, The Temporal Logic of Programs, Proceedings of the 18th Symposium on Foundation of Computer Science, IEEE, Providence, 1977, 46–57.

[29] F. A. Stomp, W. P. deRoever, Designing Distributed Algorithms by Means of Formal Sequentially Phased Reasoning, Proceedings of the 3rd International Workshop on Distributed Algorithms, LNCS 392, Springer–Verlag, 1989, 242–253.

[30] A. Valmari, Stubborn Sets for Reduced State Space Generation, 10th International Conference on Application and Theory of Petri Nets, Germany, 1989, Vol 2, 1–22.

[31] J. Zwiers, Compositionality, Concurrency and Partial Correctness, LNCS 321, Springer–Verlag, 1987.

[32] L. Zhiming , M. Joseph, Transformations of programs for fault–tolerance, to appear in Formal Aspects of Computing.