行政院國家科學委員會補助專題研究計畫成果報告

※※※※※※※※※※※※※※※※※※※※※※※※※
※ **以時間邏輯為表示法的模組細步化** ※
※ Modular Refinement in Temporal Logic ※
※ ※
※※※※※※※※※※※※※※※※※※※※※※※※※

計畫類別：☑個別型計畫　　□整合型計畫
計畫編號：NSC 89－2213－E－002－109
執行期間：89 年 8 月 1 日至 90 年 7 月 31 日
計畫主持人：蔡益坤

本成果報告包括以下應繳交之附件：
- □赴國外出差或研習心得報告一份
- □赴大陸地區出差或研習心得報告一份
- □出席國際學術會議心得報告及發表之論文各一份
- □國際合作研究計畫國外研究報告書一份

執行單位：國立臺灣大學資訊管理學系

中　華　民　國 91 **年** 4 **月** 9 **日**

行政院國家科學委員會專題研究計畫成果報告

以時間邏輯為表示法的模組細步化

Modular Refinement in Temporal Logic

計畫編號：NSC 89-2213-E-002-109
執行期間：89 年 8 月 1 日至 90 年 7 月 31 日

主持人：蔡益坤 (Yih-Kuen Tsay) 國立台灣大學資訊管理學系

中文摘要

開發大型系統時，我們往往以系統各模組的抽象規格為開端，然後將各抽象模組逐漸細步化為具體的、顯然可以實作的模組。要確保這細步化過程的正確性，我們必須能夠證明較具體模組所組成的系統確實滿足對應的較抽象模組所組成的系統的所有性質。如果有驗證法則能讓我們將這類的驗證需求化簡為驗證各相對應模組之間的關係，則整個驗證工作就會變得簡單許多。在本項研究計畫裡，我們探討如何運用 Manna 和 Pnueli 所定義的線性時間邏輯(LTL)於確保前述模組細步化過程的正確性。

Abadi 和 Lamport 這兩位研究者對以時間邏輯為表示法的模組細步化做了開創性的貢獻，他們所用的 TLA 即是時間邏輯的一種。他們的研究指出假設／保證式規格在模組細步化中扮演了基礎但隱性的角色。我們隨後以 LTL 為表示法將他們的研究結果做了一番更清楚的詮釋。本研究之初衷即在利用 LTL 表達能力的長處以獲致更易懂易用的模組細步化驗證法則。我們的主要研究成果包含了一個適用於封閉系統的模組細步化驗證法則，以及一個廣義化、適用於開放系統的驗證法則；兩者均完全以 LTL 的語法表示。

關鍵詞：假設／保證、組合式驗證、並行系統、分解、模組細步化、模組化驗證、PVS、細步化、規格、時間邏輯、驗證。

Abstract

When developing a large system, we typically start with the abstract specifications of its constituent modules and gradually refine the abstract modules into concrete, straightforwardly implementable modules. Correctness of this modular refinement process entails proof obligations of the form that the composition of more concrete modules implements the composition of more abstract modules. The verification task can be made easier if there exist proof rules that offer the possibility of breaking those proof obligations down to the module level. In this research, we investigate the use of linear-time temporal logic, specifically LTL of Manna and Pnueli, in facilitating the aforementioned modular refinement process.

Abadi and Lamport pioneered the work on modular refinement with temporal logic. They showed that assumption-guarantee specifications play a fundamental, thought implicit, role in modular refinement. We have subsequently demonstrated the advantage of LTL over TLA, the temporal logic used by Abadi and Lamport, in formulating assumption-guarantee specifications. We set to again take advantage of the expressive power of LTL so as to obtain more easily understandable and usable modular refinement rules. The main results of this work include a modular refinement rule for closed systems and a generalized one for open systems or modules; both are purely syntactical within LTL.

Keywords: Assumption-Guarantee, Compositional Verification, Concurrent Systems, Decomposition, Modular Refinement, Modular Verification, PVS, Refinement, Specification, Temporal Logic, Verification.

Modular Refinement in Temporal Logic (Final Report of NSC 89-2213-E-002-109)

Yih-Kuen Tsay
Department of Information Management
National Taiwan University
E-mail: tsay@im.ntu.edu.tw

Abstract

When developing a large system, we typically start with the abstract specifications of its constituent modules and gradually refine these abstract modules into concrete, straightforwardly implementable modules. Correctness of this modular refinement process entails proof obligations of the form that the composition of more concrete modules implements the composition of more abstract modules. The verification task can be made easier if there exist proof rules that offer the possibility of breaking those proof obligations down to the module level. In this research, we investigate the use of linear-time temporal logic, specifically LTL of Manna and Pnueli, in facilitating the aforementioned modular refinement process.

Abadi and Lamport pioneered the work on modular refinement with temporal logic. They showed that assumption-guarantee specifications play a fundamental, thought implicit, role in modular refinement. We have subsequently demonstrated the advantage of LTL over TLA, the temporal logic used by Abadi and Lamport, in formulating assumption-guarantee specifications. We set to again take advantage of the expressive power of LTL so as to obtain more easily understandable and usable modular refinement rules. The main results of this work include a modular refinement rule for closed systems and a generalized one for open systems or modules; both are purely syntactical within LTL.

Keywords: Assumption-Guarantee, Compositional Verification, Concurrent Systems, Decomposition, Modular Refinement, Modular Verification, PVS, Refinement, Specification, Temporal Logic, Verification.

1 Introduction

Large, complex systems are built from smaller, simpler modules. When developing a large system, we typically start with the abstract specifications of its constituent modules and gradually refine these abstract modules into concrete, straightforwardly implementable modules. Correctness of this modular refinement process entails proof obligations of the following form: the composition of several more concrete modules implements the composition of their corresponding more abstract modules. The verification task can be made easier if there exist proof rules

that offer the possibility of breaking those proof obligations down to the module level. As a more concrete module has steps with finer granularity of atomicity and hence is more easily interfered by its environment (i.e., other modules), the implementation relationship between the module and its corresponding abstract module in general does not hold. What one probably can hope for is that the implementation relationship will hold with certain assumption about the environment of the more concrete module and the assumption can later be discharged with only the abstract modules taken into consideration (otherwise, there would be no simplification).

In this research, we investigate the use of linear-time temporal logic in facilitating the aforementioned modular refinement process. Temporal logic is one convenient formalism for specifying and reasoning about the behaviors of a system. The idea of representing concurrent systems and their specifications as temporal-logic formulas was first proposed by Pnueli [Pnu82]. Fundamental concepts in formal verification such as refinement (implementation), hiding, and parallel composition can all be conveniently treated with the logic [Lam94, MP92, AL95]. There are variations of temporal logic. We shall adopt the linear-time temporal logic of Manna and Pnueli [MP92], which we refer to as LTL.

Abadi and Lamport pioneered the work on modular refinement (which they referred to as decomposition of specifications) with temporal logic [AL95]. They showed that assumption-guarantee specifications play a fundamental, though implicit, role in modular refinement. We have subsequently demonstrated the advantage of LTL over TLA [Lam94], the temporal logic used by Abadi and Lamport, in formulating assumption-guarantee specifications [JT96]. We set to again take advantage of the expressive power of LTL so as to obtain more easily understandable and usable modular refinement rules. Below is a brief account of recent works that are most related to this work.

- In [AL95], Abadi and Lamport proposed a way of writing assumption-guarantee specifications in TLA [Lam94] and presented a general rule for composing such specifications. However, they used the notion of safety closure (which is a semantic concept) in the rule. Two decomposition (modular refinement) rules were derived from the general composition rule.

- Jointly with Jonsson, we showed in [JT96] how to write and reason about assumption-guarantee specifications in LTL. Unlike those in Abadi and Lamport [AL95], the composition rules in this work are purely syntactical within LTL.

- Kurshan and Lamport [KL93] demonstrated the use of a decomposition rule in breaking the verification of a multiplier down to the verification of its individual components, which can be carried out by model checking. Theorem proving is used to verify that the complete system satisfies its specification if each of its components does.

- In a more recent work of ours [Tsa00], we consider two forms of temporal formulas that

correspond to the strong and the weak interpretations of an assumption-guarantee specification and investigate how they can be applied in compositional (modular) verification. We argue by examples that the two forms complement each other and both are needed to facilitate the compositional approach.

The main results of this work include a modular refinement rule for closed systems and a generalized one for open systems or modules; both are purely syntactical within LTL. In the next section, we give the necessary preliminaries of temporal logic. Section 3 explains the intricacy of simplifying correctness proofs of modular refinement. The main results can be found in Section 4.

2 Temporal Logic and Specification

2.1 LTL

LTL, the linear-time temporal logic of Manna and Pnueli [MP92], is a logic for expressing and reasoning about properties of infinite sequences of states, where each state is an assignment to a predefined set of variables. The language of LTL assumes a set of constant, function, and predicate symbols with fixed interpretations. It classifies each variable as being *rigid*—having the same interpretation in all states of a sequence or *flexible*—with no restrictions on interpretation in different states; flexible variables are typically used for representing program or control variables, whose value may change over time. Primitive temporal formulas in LTL, called *state formulas*, are built from variables, constants, functions, and predicates using the usual first-order logical connectives. A state formula is interpreted over a state where each variable in the formula is assigned a value; this is analogous to first-order logic.

The expressive power of LTL mainly comes from *temporal operators*. In this paper, we will explicitly use only three temporal operators: $\Box$, $\ominus$, and $\boxminus$. A general *temporal formula* is constructed by applying temporal operators and first-order logical connectives to state formulas. An LTL temporal formula is interpreted over an infinite sequence of states, relative to a position in that sequence. We give below the semantics for temporal formulas involving a quantifier or one of the three temporal operators:

- $(\sigma, i) \models \Box\varphi$ iff $\forall k \geq i : (\sigma, k) \models \varphi$. In other words, $\Box\varphi$ (read as "henceforth φ" or "always φ") holds at a position if φ holds at that current and all following positions.
- $(\sigma, i) \models \ominus\varphi$ iff $(i > 0) \rightarrow ((\sigma, i-1) \models \varphi)$. In other words, $\ominus\varphi$ (read as "before φ") holds at position i if either position i is the first position (i.e., $i = 0$) of the sequence or φ holds at position $i - 1$.

- $(\sigma, i) \models \boxminus\varphi$ iff $\forall k : 0 \leq k \leq i : (\sigma, k) \models \varphi$. In other words, $\boxminus\varphi$ (read as "so-far φ") holds at a position if φ holds at that position and all preceding positions;

A sequence σ' is called a *u-variant* of σ if σ' differs from σ in at most the interpretation given to u in each state; note that the restrictions of rigid variables must be observed.

- $(\sigma, i) \models \exists u : \varphi$ iff $(\sigma', i) \models \varphi$ for some *u-variant* σ' of σ. Intuitively, this means that the truth of $\exists u : \varphi$ for a flexible variable u depends on the existence of an infinite sequence of u-values (one for each state), rather than just a single value, such that φ can be satisfied.
- $(\sigma, i) \models \forall u : \varphi$ iff $(\sigma', i) \models \varphi$ for every *u-variant* σ' of σ.

We say that a sequence σ satisfies a formula φ if $(\sigma, 0) \models \varphi$, often abbreviated as $\sigma \models \varphi$. A formula φ is *valid*, denoted $\models \varphi$, if φ is satisfied by every sequence.

When using LTL to specify a system, one very often would want to express constraints on the state transitions that the system can make. This requires a means for specifying the values of an expression in any two consecutive states (a position and its preceding position) of a sequence. As one possible solution, Manna and Pnueli introduced the notation u^- for denoting the value of u in the preceding state. Formally, the interpretation of u^- in position i is the same as the interpretation of variable u in position $i-1$; by convention, the interpretation of u^- in position 0 is the same as the interpretation of u in position 0. Note that the new notation does not increase the expressive power of LTL, as any formula with "$^-$"-superscribed variables can be translated into an equivalent formula without such variables. A formula without temporal operators but possibly with "$^-$"-superscribed variables is called a *transition formula* (this definition is slightly different from that in [MP92]).

2.2 Specifying a System

A system consists of a set of variables, an initial condition on the variables, and a set of transitions that specify how the system may change the values of its variables in an execution step. Semantically, a system is associated with a set of computations (behaviors) or sequences of states, each of which represents a possible execution of the system. Such a system is amenable to specification with temporal formulas.

We find it convenient to distinguish two kinds of specification: system specification and requirement specification. The system specification of a system is a temporal formula that characterizes exactly the set of all possible behaviors of the system, while a requirement specification gives some superset of this set. We will talk mostly of system specifications.

System specifications are essentially programs in the form of a temporal formula. Consider Program GCD shown in Figure 1. The system specification of GCD can be given as follows.

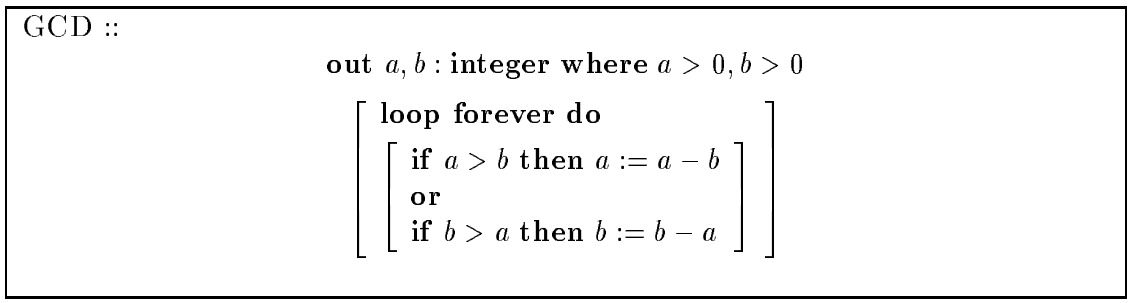

Figure 1: Program GCD.

$$M_{\mathrm{GCD}} \triangleq (a > 0) \wedge (b > 0) \wedge \Box \left(\begin{array}{ll} & (a^- > b^-) \wedge (a = a^- - b^-) \wedge (b = b^-) \\ \vee & (b^- > a^-) \wedge (b = b^- - a^-) \wedge (a = a^-) \\ \vee & (a = a^-) \wedge (b = b^-) \end{array} \right)$$

The formula M_{GCD} states that initially the values of a and b are greater than 0. It also states via the disjunction of three transition formulas that, in each step of an execution, either (1) the value of a is decremented by that of b (while the value of b remains unchanged) when the value of a is greater than that of b, (2) the value of b is decremented by that of a (while the value of a remains unchanged) when the value of b is greater than that of a, or (3) nothing is changed. The transition formula $(a = a^-) \wedge (b = b^-)$ is called a stuttering transition and is included to make the specification invariant under stuttering. This stuttering transition also helps make the specification simpler. It holds at position 0 of every execution sequence of GCD. Without it, the other two transition formulas would have to be modified so that they also hold at position 0.

We will regard system specifications as formal definitions of systems so that we can do without a formal syntax and semantics of the programming language. Programs are informal notations for readability.

2.3 Common Forms of Formulas

A formula without any future operator ($\Box$ is the only future operator we explicitly use in this paper) is called a *past formula*; in particular, a transition formula is a past formula. A *safety formula* is one that specifies a safety property and a *liveness formula* is one that specifies a liveness property[1]. Of particular importance, formulas of the form $\Box H$ are safety formulas if the truth value of H depends only on the present and the past states, e.g., if H is a past formula. The form of a liveness formula is not important for our purposes.

[1] A property is said to be a safety property if the following condition holds: for any sequence σ, if each prefix of σ is a prefix of some sequence that satisfies the property, then σ also satisfies the property; and, a property is said to be a liveness property if every finite sequence is a prefix of some sequence that satisfies the property [AS85].

The safety formula in a system specification can be put in the canonical form of $\Box H$, specifically in the form of $\Box((\mathit{first} \wedge \mathit{Init}) \vee (\neg \mathit{first} \wedge N))$, where *first* abbreviates $\ominus \mathit{false}$ which holds only at position 0 of a sequence, *Init* is a state formula, and N the disjunction of several transition formulas, including a stuttering transition. As N will always contain a stuttering transition, $\Box((\mathit{first} \wedge \mathit{Init}) \vee (\neg \mathit{first} \wedge N))$ simplifies to $\Box((\mathit{first} \wedge \mathit{Init}) \vee N)$. For example, the system specification of GCD can be equivalently written as follows.

$$M_{\mathrm{GCD}} \triangleq \Box \left(\begin{array}{ll} & \mathit{first} \wedge (a > 0) \wedge (b > 0) \\ \vee & (a^- > b^-) \wedge (a = a^- - b^-) \wedge (b = b^-) \\ \vee & (b^- > a^-) \wedge (b = b^- - a^-) \wedge (a = a^-) \\ \vee & (a = a^-) \wedge (b = b^-) \end{array} \right)$$

More generally, the system specification of a system can be put in the following form:

$$\exists x : \Box H \wedge L$$

where x is a tuple of flexible variables, H is a past formula (so that $\Box H$ is a safety formula), and L is a liveness formula. Formulas of this form are referred to as *canonical formulas*.

It is desirable that the pair of $\Box H$ and L be "*machine-closed*" [AL91], i.e., the safety closure[2] $\mathcal{C}(\Box H \wedge L)$ of $\Box H \wedge L$ is equivalent to $\Box H$. One way of understanding machine-closedness is that L does not rule out safety properties that are allowed by $\Box H$.

If, in addition, $\Box H$ is stuttering-extensible, then $\mathcal{C}(\exists x : \Box H \wedge L)$ can be shown to equal $\Box(\exists x : \boxminus H)$ [JT96]. A safety formula is said to be stuttering-extensible if a system that has satisfied the formula so far, then it will continue to satisfy the formula simply by doing nothing. When $\Box H$ is of the form $\Box((\mathit{first} \wedge \mathit{Init}) \vee N)$ where N includes a stuttering transition, it is guaranteed to be stuttering-extensible.

3 Modular Specification and Refinement

3.1 Composition as Conjunction

Program GCD can be decomposed as the parallel composition of two modules, shown in Figure 2.

A module may read but not change the value of an **in** (input) variable. A *compatible* environment of a module may read but not change the value of an **own out** (owned output) variable of the module. In the system $\Pi_a \,\|\, \Pi_b$, Π_b is the environment of Π_a and Π_a is the environment of Π_b; both are clearly compatible with each other. Computations of a module are the sequences

[2] The *safety closure* of a given property is the strongest safety property implied by the given property, i.e., a safety property satisfied by exactly those sequences σ such that each prefix of σ is a prefix of some sequence that satisfies the given property. The safety closure of a property specified by an arbitrary temporal formula can also be expressed as a temporal formula, making it meaningful to talk about the safety closure of a formula in LTL [JT96].

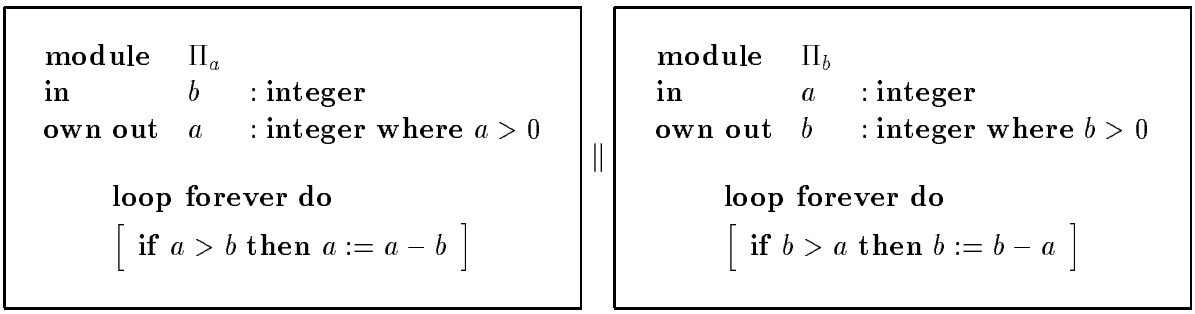

Figure 2: Program GCD as the parallel composition of two modules.

of states produced when the module is executed in parallel with an arbitrary but compatible environment, i.e., the computations of an imaginary system obtained from composing the module with an arbitrary environment. A module satisfies a certain property if the corresponding assertion holds for each of its computations.

The system specifications M_a and M_b of modules Π_a and Π_b respectively are as follows:

$$M_a \triangleq \Box \left(\begin{array}{ll} & \mathit{first} \wedge (a = 0) \\ \vee & (a^- > b^-) \wedge (a = a^- - b^-) \wedge (b = b^-) \\ \vee & (a = a^-) \end{array} \right)$$

$$M_b \triangleq \Box \left(\begin{array}{ll} & \mathit{first} \wedge (b = 0) \\ \vee & (b^- > a^-) \wedge (b = b^- - b^-) \wedge (a = a^-) \\ \vee & (b = b^-) \end{array} \right)$$

It is perhaps more accurate to say that M_a is the system specification of an imaginary system composed of Π_a and an arbitrary but compatible environment; same for M_b.

3.2 Modular Refinement

We decompose a system into smaller modules so as to develop or, to be formally correct, *refine* each module separately. Suppose M_i is the specification of the i-th of n modules of a system and M_i^l that of the corresponding refined module. We are faced with the following proof obligation:

$$\bigwedge_{i=1}^{n} M_i^l \rightarrow \bigwedge_{i=1}^{n} M_i$$

Ideally, we would like to establish the validity of the above implementation relation by proving that $M_i^l \rightarrow M_i$ for each i. But, this is usually not possible.

Consider the GCD program decomposed as the parallel composition of Π_a and Π_b. We may want to remove the simultaneous atomic accesses to a and b by refining the modules as shown in Figure 3, where atomicity is made explicit with angle brackets.

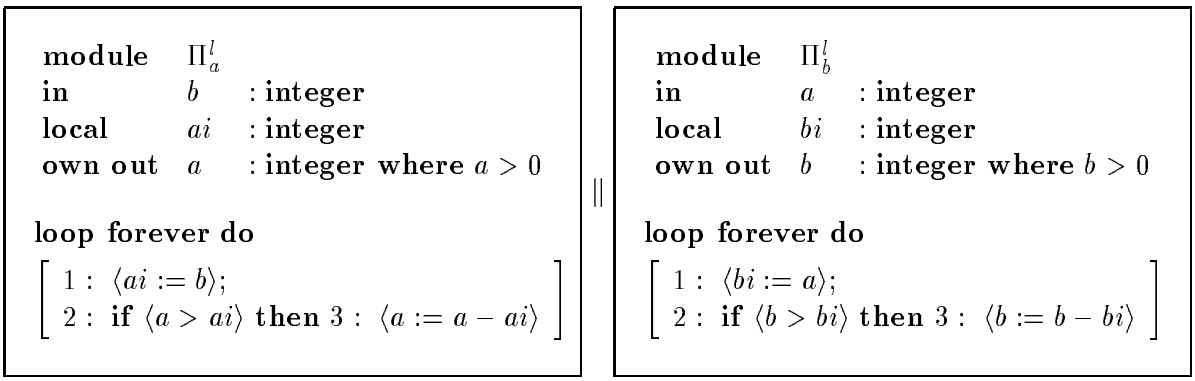

Figure 3: Program GCD as the parallel composition of two refined modules.

The system specifications M_a^l and M_b^l of modules Π_a^l and Π_b^l respectively are as follows:

$$M_a^l \triangleq \exists ai, pca : \Box H_a^l$$

$$M_b^l \triangleq \exists bi, pcb : \Box H_b^l$$

where

$$\begin{array}{rcl} H_a^l & \triangleq & \begin{array}{ll} & \mathit{first} \wedge (a > 0) \wedge (pca = 1) \\ \vee & (pca^- = 1) \wedge (ai = b^-) \wedge (pca = 2) \wedge (a = a^-) \wedge (b = b^-) \\ \vee & (pca^- = 2) \wedge (a^- > ai^-) \wedge (pca = 3) \wedge (ai = ai^-) \wedge (a = a^-) \wedge (b = b^-) \\ \vee & (pca^- = 2) \wedge \neg(a^- > ai^-) \wedge (pca = 1) \wedge (ai = ai^-) \wedge (a = a^-) \wedge (b = b^-) \\ \vee & (pca^- = 3) \wedge (a = a^- - ai^-) \wedge (pca = 1) \wedge (ai = ai^-) \wedge (b = b^-) \\ \vee & (pca = pca^-) \wedge (ai = ai^-) \wedge (a = a^-) \end{array} \end{array}$$

$$\begin{array}{rcl} H_b^l & \triangleq & \begin{array}{ll} & \mathit{first} \wedge (b > 0) \wedge (pcb = 1) \\ \vee & (pcb^- = 1) \wedge (bi = a^-) \wedge (pcb = 2) \wedge (b = b^-) \wedge (a = a^-) \\ \vee & (pcb^- = 2) \wedge (b^- > bi^-) \wedge (pcb = 3) \wedge (bi = bi^-) \wedge (b = b^-) \wedge (a = a^-) \\ \vee & (pcb^- = 2) \wedge \neg(b^- > bi^-) \wedge (pcb = 1) \wedge (bi = bi^-) \wedge (b = b^-) \wedge (a = a^-) \\ \vee & (pcb^- = 3) \wedge (b = b^- - bi^-) \wedge (pcb = 1) \wedge (bi = bi^-) \wedge (a = a^-) \\ \vee & (pcb = pcb^-) \wedge (bi = bi^-) \wedge (b = b^-) \end{array} \end{array}$$

Although the composition of Π_a^l and Π_b^l correctly implements the composition of Π_a and Π_b, i.e., $M_a^l \wedge M_b^l \rightarrow M_a \wedge M_b$, neither $M_a^l \rightarrow M_a$ nor $M_b^l \rightarrow M_b$ holds in isolation. Π_a^l does not implement Π_a because Π_a^l can behave in ways that Π_a cannot. When Π_a^l is run with an *arbitrary but compatible* environment, the value of b may be changed before the assignment "$a := a - ai$" is executed, resulting in a being decremented by a previous value of b. Analogously, Π_b^l does not implement Π_b.

However, Π_a^l is not intended to be run with just any environment. If Π_a^l is run with an environment where the value of b may be changed only when $a < b$, then in this context Π_a^l indeed

correctly implements Π_a. Let E_a represent the needed constraint on the environment. The contextual implementation relation is expressed as $E_a \wedge M_a^l \rightarrow M_a$. In summary, we wish to conclude $M_a^l \wedge M_b^l \rightarrow M_a \wedge M_b$ from roughly the following premises.

$$\begin{array}{l} E_a \wedge M_a^l \rightarrow M_a \\ E_b \wedge M_b^l \rightarrow M_b \\ M_a \wedge M_b \rightarrow E_a \\ M_a \wedge M_b \rightarrow E_b \end{array}$$

Each of the premises is simpler than the original proof obligation. The assumptions E_a and E_b on the environments of Π_a^l and Π_b^l respectively needed in the first two premises are discharged by the last two premises, which in turns need M_a and M_b. Discharging E_a and E_b in this way is a circular reasoning and, in general, unsound. The first modular refinement rule in the next section tells what the actual premises should be.

4 Rules for Modular Refinement

We present two proof rules that facilitate modular refinement. The first is for closed systems and the second is a generalized rule for open systems (or modules).

In light of the results in [JT96], we consider specifications expressed in the canonical form with the additional conditions of machine-closedness and stuttering-extensibility. Specifically, the specification M of a module is expressed as $\exists y : \Box H_M \wedge L_M$ where $\mathcal{C}(\Box H_M \wedge L_M) \leftrightarrow \Box H_M$ and $\Box H_M$ is stuttering-extensible so that the safety closure of M is equivalent to $\Box(\exists y \colon \boxminus H_M)$. The assumption E about an environment is expressed as $\Box(\exists x : \boxminus H_E)$. Assuming that $\Box H_E$ is stuttering-extensible, $\Box(\exists x : \boxminus H_E)$ expresses the safety closure of $\exists x : \Box H_E$, which is a specification of the safety properties of a system with the tuple x of internal variables hidden ($\exists x : \Box H_E$ is not necessarily a safety formula.) If no internal variables are used, E becomes $\Box \boxminus H_E$, or simply $\Box H_E$.

Theorem 4.1 *Assuming that x_1, ..., x_n, y_1, ..., y_n, y_1^l, ..., y_n^l are pairwise disjoint tuples of variables and no free variables become bound,*

$$\begin{array}{lll} 1. & & \models \Box\Big[(\exists y_1 \ldots y_n \colon \boxminus \bigwedge_{j=1}^{n} H_{M_j}) \rightarrow (\exists x_i \colon \boxminus H_{E_i})\Big], \ \textit{for } 1 \leq i \leq n \\ 2. & (a) & \models \Box\Big[\circledcirc(\exists x_i \colon \boxminus H_{E_i}) \wedge (\exists y_i^l \colon \boxminus H_{M_i^l}) \rightarrow (\exists y_i \colon \boxminus H_{M_i})\Big], \ \textit{for } 1 \leq i \leq n \\ & (b) & \models \Box(\exists x_i \colon \boxminus H_{E_i}) \wedge M_i^l \rightarrow M_i, \ \textit{for } 1 \leq i \leq n \\ \hline & & \models \bigwedge_{i=1}^{n} M_i^l \rightarrow \bigwedge_{i=1}^{n} M_i \end{array}$$

Premise 2 requires reasoning about only one module at a time. Assumption $\Box(\exists x_i : \boxminus H_{E_i})$ about the environment of M_i^l needs to be discharged in Premise 1, which requires reasoning

about $\bigwedge_{i=1}^{n} M_i$, but this conjunction is more abstract and usually simpler than $\bigwedge_{i=1}^{n} M_i^l$. To apply the rule to the refinement of GCD, we formulate the assumptions E_a ($= \Box H_{E_a}$) and E_b ($= \Box H_{E_b}$) as follows.

$$\begin{array}{lcl} \Box H_{E_a} & \triangleq & \Box(((a^- < b^-) \wedge (a = a^-)) \vee (b = b^-)) \\ \Box H_{E_b} & \triangleq & \Box(((a^- > b^-) \wedge (b = b^-)) \vee (a = a^-)) \end{array}$$

The next rule is a generalized rule for dealing with open systems (or modules); closed systems are a special case of open systems that do not interact with the environment. It is needed if one wishes to establish a refinement relationship inductively (as opposed to doing it in one shot using the first rule).

Theorem 4.2 *Assuming that x, x_1, ..., x_n, y_1, ..., y_n, y_1^l, ..., y_n^l are pairwise disjoint tuples of variables and no free variables become bound,*

$$\begin{array}{lll} 1. & & \models \Box\Big[(\exists x : \boxminus H_E) \wedge (\exists y_1 \ldots y_n : \boxminus \bigwedge_{j=1}^{n} H_{M_j}) \rightarrow (\exists x_i : \boxminus H_{E_i})\Big], \text{ for } 1 \leq i \leq n \\ 2. & (a) & \models \Box\Big[\circleddash(\exists x_i : \boxminus H_{E_i}) \wedge (\exists y_i^l : \boxminus H_{M_i^l}) \rightarrow (\exists y_i : \boxminus H_{M_i})\Big], \text{ for } 1 \leq i \leq n \\ & (b) & \models \Box(\exists x_i : \boxminus H_{E_i}) \wedge M_i^l \rightarrow M_i, \text{ for } 1 \leq i \leq n \\ \hline & (a) & \models \Box\Big[\circleddash(\exists x : \boxminus H_E) \wedge (\exists y_1^l \ldots y_n^l : \boxminus \bigwedge_{i=1}^{n} H_{M_i^l}) \rightarrow (\exists y_1 \ldots y_n : \boxminus \bigwedge_{i=1}^{n} H_{M_i})\Big] \\ & (b) & \models \Box(\exists x : \boxminus H_E) \wedge \bigwedge_{i=1}^{n} M_i^l \rightarrow \bigwedge_{i=1}^{n} M_i \end{array}$$

5 Concluding Remarks

This work was inspired by Abadi and Lamport [AL95] and closely followed the development of their work. We have used the same example, namely the GCD program, to illustrate the intricacy of simplifying correctness proofs of modular refinement. Modular refinement rules were called decomposition theorems in [AL95]. Unlike those in [AL95], the decomposition rules in this work are purely syntactical within LTL, which we consider the main contribution of this work.

We are currently working out a mechanization of these modular refinement rules in PVS [COR+95].

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 24(4):181–185, October 1985.

[COR+95] J. Crow, S. Owre, J.M. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1995.

[JT96] B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167:47–72, October 1996. An extended abstract appeared earlier in TAPSOFT '95, LNCS 915.

[KL93] R.P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In C. Courcoubetis, editor, *Computer-Aided Verification, CAV '93, LNCS 697*, pages 166–179. Springer-Verlag, June 1993.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[Pnu82] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1982.

[Tsa00] Y.-K. Tsay. Compositional verification in linear-time temporal logic. In J. Tiuryn, editor, *Proceedings of the Third International Conference on Foundations of Software Science and Computation Structures, LNCS 1784*, pages 344–358. Springer, March 2000.