行政院國家科學委員會專題研究計畫　成果報告

元件合成軟體之形式化驗證初探

計畫類別：個別型計畫
計畫編號：NSC92-2213-E-002-061-
執行期間：92 年 08 月 01 日至 93 年 07 月 31 日
執行單位：國立臺灣大學資訊管理學系暨研究所

計畫主持人：蔡益坤

報告類型：精簡報告
報告附件：出席國際會議研究心得報告及發表論文
處理方式：本計畫可公開查詢

中　華　民　國 94 年 4 月 13 日

行政院國家科學委員會專題研究計畫成果報告

元件合成軟體之形式化驗證初探

Toward Formal Verification of Component-Based Software

計畫編號：NSC 92-2213-E-002-061

執行期間：92 年 8 月 1 日至 93 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

摘要

以元件為基礎之設計已成為軟體開發的主流方法之一。本研究旨在探討元件合成軟體正確性規格與驗證的問題。由於構成這類軟體的各元件通常是由不同團體獨立開發完成，元件設計者需要一種描述元件而不揭露其程式碼的方法，也就是一種可描述元件做什麼而不必交代它是如何做的方法。而元件合成軟體開發者需要一套驗證方法，在給定他個人所寫的程式碼以及所引用元件的規格後，檢驗他的合成系統確實正確無誤。

過去我們已發展出一套組合式規格與驗證的架構，本研究嘗試將這套架構運用在前述元件合成軟體規格與驗證的工作。我們特別針對部份元件僅有規格而無程式碼的情形，闡述這套架構如何使用。我們同時也探討如何以自動化驗證工具來協助這些驗證工作的進行。

關鍵詞：假設／保證、元件合成軟體、組合式驗證、形式化方法、模型檢驗器、模組化驗證、證明助理、軟體元件、規格、時間邏輯、驗證。

Abstract

Component-based design has become a prominent approach to software development. In this project we consider the task of formally specifying and verifying behavioral correctness of software built from independently developed components. The task requires a way for the component designer to describe his component without revealing the code, i.e., to specify what the component does but not how it gets things done. It also requires a way for the component-based software developer to verify his system, given the source code of the part that he coded and the specifications of the components that he used.

We adapt a compositional framework that we have developed earlier for carrying out the aforementioned specification and verification tasks. In particular, we demonstrate how the framework may be used in a typical case where some components of a system can be reasoned only with their behavioral specification but not their code. We also investigate how the verification tasks can be facilitated by automated tools.

Keywords: Assumption-Guarantee, Component-Based Software, Compositional Verification, Formal Methods, Model Checkers, Modular Verification, Proof Assistants, Software Components, Specification, Temporal Logic, Verification.

Toward Formal Verification of Component-Based Software
元件合成軟體之形式化驗證初探
(Final Report of NSC 92-2213-E-002-061)

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

Abstract

Component-based design has become a prominent approach to software development. In this project we consider the task of formally specifying and verifying behavioral correctness of software built from independently developed components. The task requires a way for the component designer to describe his component without revealing the code, i.e., to specify what the component does but not how it gets things done. It also requires a way for the component-based software developer to verify his system, given the source code of the part that he coded and the specifications of the components that he used.

We adapt a compositional framework that we have developed earlier for carrying out the aforementioned specification and verification tasks. In particular, we demonstrate how the framework may be used in a typical case where some components of a system can be reasoned only with their behavioral specification but not their code. We also investigate how the verification tasks can be facilitated by automated tools.

Keywords: Assumption-Guarantee, Component-Based Software, Compositional Verification, Formal Methods, Model Checkers, Modular Verification, Proof Assistants, Software Components, Specification, Temporal Logic, Verification.

1 Introduction

Contemporary software development practices emphasize reuse of components. Composite systems composed of software components are often referred to as component-based software, or simply component software. In [19], Szyperski gives the following definition for a software component:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Software components encapsulate collections of data and operations as integral functioning units and, with well-defined interfaces, are *reusable*. Reusability of software components makes

it possible for any component in a component-based software system to be replaced by a functionally equivalent (or superior) component from a different vendor. As this may considerably reduce development cost and time, component-based design has become a prominent approach to software development.

Component-based software systems may be structured in many different ways. We will focus on *concurrent* systems that are a *parallel* composition of concurrent software components. The correctness of such software relies not only on the static aspects such as types of input/output variables *but also on the dynamic aspects such as temporal behaviors of the components.* Practical development tools support only the static aspects.

In this project we consider the task of formally specifying and verifying *behavioral* correctness of software built from independently developed components. The task requires a way for the component designer to describe his component without revealing the code, i.e., to specify what the component does but not how it gets things done. The specification should be sufficiently precise so that some other system can make a correct usage of the component in that, with the component as a part, the system as a whole behaves properly. It also requires a way for the component-based software developer to verify his system, given the source code of the part that he programs and the specifications of the components that he uses.

A specification and verification framework for component-based software should at least consist of a formalism for specifying software components and a mechanism for composing such specifications. The formalism for component specifications dictates the composition mechanism. Properties of a concurrent system may be represented by assertions on computations of the system and so may properties of a concurrent component or module. Computations of a system are the sequences of states produced when the system is executed in isolation. In contrast, computations of a module are the sequences of states produced when the module is executed in parallel with an arbitrary but (syntactically) compatible environment, i.e., the computations of an imaginary system obtained from composing the module with the arbitrary environment. A system or module satisfies a certain property if the corresponding assertion holds for each of its computations.

A module is meant for particular contexts or environments and will behave properly only if its environment does. When specifying properties of a module, one should therefore include (1) assumed properties about its environment and (2) guaranteed properties of the module if the environment obeys the assumption. This type of specification is essentially a generalization of pre and post-conditions for sequential programs [9]. The generalization was adopted in the early 1980's by Misra and Chandy [18], Jones [10], and Lamport [14] and became the so-called *assumption-guarantee* (also known as rely-guarantee or assumption-commitment) paradigm.

We have developed in [13, 21] a formulation of assumption-guarantee specifications that are suitable for specifying the modules of a concurrent system, along with proof rules for composing such specifications. Our compositional framework is formulated in the linear temporal logic of Manna and Pnueli [17], which is referred to as LTL here. LTL formulas express properties of

an infinite computation and are suitable for specifying concurrent systems or modules. In more recent work [20], we adapt the compositional framework so as to make it more convenient for mechanization. The adapted method has been mechanized in the well-used verification tool PVS [6], a proof assistant based on higher-order logic.

The primary goal of this research is to test using our compositional framework in the aforementioned specification and verification tasks for component-based software. This is part of our continuous effort in extending the line of research on applications of LTL. In particular, we demonstrate how the framework may be used in a typical case where some components of a system can be reasoned only with their behavioral specification but not their code. We also investigate how the verification tasks can be facilitated by automated tools.

Note: This research represents our initial investigation on the concerned subject and is of foundational nature. We shall treat software mainly in the semantic level, modelling them as state-transition systems.

2 Related Work

Over the past few years, significant progresses have been made in the development of compositional methods, some for general purposes and some tailored to component-based software design. Many of these works follow the assumption-guarantee paradigm. Below is a partial list of the more notable recent ones.

- In a more recent work [8], de Alfaro and Henzinger proposed an automata-based language called *interface automata* that is motivated by the formal validation of component-based design. This work can be related to recent trends in capturing behavioral aspects of software by type systems in the sense that interface compatibility checking is done at compile time like type checking. Interface automata interact with one another through the synchronization of input and output events; local events are interleaved. An interface automaton may accept only some input behaviors generated by other automata that represent its environment. With such a mechanism, an interface automaton can capture both assumptions about the order in which the methods of a component are called, and guarantees about the order in which the component calls external methods, which are limited yet practically useful temporal aspects of software component interfaces.

  Though syntactically similar to two earlier compositional modelling formalisms I/O automata [16, 15] and I/O systems [11, 12], interface automata are not required to be receptive (input-enabled). Whether two components are compatible, i.e., whether they can be used together in some environment, can be formally verified by checking if the composition of their representing interface automata permits some environment that never leads the composite automaton into an error state.

- Chakrabarti *et al.* [4] extended the preceding formalism to support recursive call-backs

between modules. In [5], the same group of people went further with the generalization effort to define Moore interfaces and bidirectional interfaces, which have the additional expressive power of modelling the behavior of a system component. Bidirectional interfaces may dynamically change the role (input or output) of a variable to model components that have bidirectional connections.

- In a follow-up work [7] of Interface Automata, de Alfaro argued that composition and refinement of open systems should be phrased in game-theoretical terms. For instance, the interface compatibility checking problem is equivalent to solving a two-player game between the composite automaton (which tries to get to an error state) and the environment (which tries to prevent it).
- Alur *et al.* [2, 1] introduced the notion of modular strategies for games on recursive state machines. A modular strategy has only local memory and does not rely the context in which the module is invoked. They maintained that checking for the existence of modular strategies matches better with the intuition for interface compatibility.
- In a follow-up work, Alur *et al.* [3] proposed an algorithm for extracting the interface specification of a Java class from its code. The interface specification prescribes the correct sequencing of method invocations to the Java class, which may be useful for developers using the class. The work follows the game-theoretical approach to describing and verifying behavioral correctness of program modules.

3 Modular Verification in LTL

We briefly review the temporal-logic framework developed by Jonsson and Tsay [13, 21] for compositional specification and verification. Compositional verification hinges on an effective formulation for specifying the modules of a system. A key ingredient of their framework is the definition of assumption-guarantee (A-G) formulas for specifying properties of a module. Two types of A-G formulas had been defined. We consider only *strong* A-G formulas, referred to simply as A-G formulas. A-G formulas have a mutual induction mechanism built in and can be more readily composed.

We omit a detailed description of LTL and simply recall that $\Box$ means "always in the future (including the present)"; $\circleddash$ means "in the previous state (if there is any)"; $\boxminus$ means "always in the past (including the present)".)

Assuming that the assumption and the guarantee are canonical safety formulas respectively of the forms $\Box H_A$ and $\Box H_G$ (where H_A and H_G are past formulas), an A-G formula $\Box H_A \rhd \Box H_G$ is defined as follows:

$$\Box H_A \rhd \Box H_G \triangleq \Box(\circleddash \boxminus H_A \to H_G).$$

The formula $\Box(\circleddash \boxminus H_A \to H_G)$, which is equivalent to $\Box(\circleddash \boxminus H_A \to \boxminus H_G)$, essentially says that $\boxminus H_G$ holds at least one step longer than $\boxminus H_A$ does; in particular, it asserts that H_G holds

initially. Suppose that H_{G_1} and H_{G_2} are past formulas. Then,

$$\models \; (\Box H_{G_1} \rhd \Box H_{G_2}) \wedge (\Box H_{G_2} \rhd \Box H_{G_1}) \;\rightarrow\; \Box H_{G_1} \wedge \Box H_{G_2}.$$

The above result is essentially the composition principle formulated by Misra and Chandy [18]. It illustrates that A-G formulas have a mutual induction mechanism built in and hence permit "circular reasoning" (there is of course no real cycle if one looks at the semantic models and reasons state by state from the initial one), i.e., deducing new properties from mutually dependent properties. Below is a more general rule for composing A-G specifications:

Theorem 3.1 *Suppose that $\Box H_{A_i}$ and $\Box H_{G_i}$, for $1 \leq i \leq n$, $\Box H_A$, and $\Box H_G$ are canonical formulas. Then,*

$$\frac{\begin{array}{ll} 1. & \models \; \Box\Big(\boxminus H_A \wedge \boxminus \bigwedge_{i=1}^{n} H_{G_i} \rightarrow H_{A_j} \Big), \; \text{for } 1 \leq j \leq n \\ 2. & \models \; \Box\Big(\circleddash \boxminus H_A \wedge \boxminus \bigwedge_{i=1}^{n} H_{G_i} \rightarrow H_G \Big) \end{array}}{\models \; \bigwedge_{i=1}^{n} (\Box H_{A_i} \rhd \Box H_{G_i}) \;\rightarrow\; (\Box H_A \rhd \Box H_G)}$$

Intuitively, Premise 1 of the above composition rule says that the assumption about the environment of a module should follow from the guarantees of other modules and the assumption about the environment of the entire system (which may in turn be a larger module), while Premise 2 says that the guarantee of the entire system should follow from the guarantees of individual modules and the assumption about its environment. We will not consider liveness properties in this project, as the built-in mutual induction mechanism (which is most characteristic of an A-G formula) really works for safety properties only.

Mechanization of this compositional framework is not as straightforward as it might appear. In the definition of an A-G formula, the assumption A and the guarantee G are assumed to be given respectively as $\Box H_A$ and $\Box H_G$, where H_A and H_G are past formulas. The assumption leads to a more succinct formulation of A-G specifications and rules for composing such specifications. Unfortunately, in a general-purpose proof assistant it is inconvenient to enforce the restriction of a temporal formula being a past one. For the ease of mechanization, we restrict assumptions and guarantees to be of the canonical form $\Box((\mathit{first} \rightarrow \mathit{Init}) \wedge N)$ ($\equiv \mathit{Init} \wedge \Box N$), where N is an action formula that relates the current state and the previous state of the system. This restriction does lose much expressiveness, as canonical safety formulas are typical for specifying safety properties of a system.

Now we consider hiding of local variables. Hiding is a common technique for making specifications more abstract and corresponds to existential quantification over flexible variables in LTL. Jonsson and Tsay [13] also considered A-G specifications where the assumption and the guarantee parts involve hiding. We will summarize the relevant part of their work below but with a slightly different style of presentation. In particular, we make the free variables of a formula explicit and we also incorporate a simplification to the definition of an A-G formula. A temporal formula φ may be written as $\varphi(z)$ or $\varphi(z, x)$ to indicate that the free (flexible)

variables of φ are among the tuple z or the tuples z, x of variables. We write $\exists x\colon \varphi(z,x)$ to hide the x part of variables of a formula $\varphi(z,x)$.

An A-G formula with assumption $\Box(\exists x\colon \boxminus H_A(z,x))$ and guarantee $\Box(\exists y\colon \boxminus H_G(z,y))$ is defined below. The formula $\Box(\exists x\colon \boxminus H_A(z,x))$ represents the "safety part" (formally, safety closure) of $\exists x\colon \Box H_A(z,x)$ when H_A includes a stuttering step. The formula $\exists x\colon \Box H_A(z,x)$ may not be a safety formula, though $\Box H_A(z,x)$ is a safety one.

$$\Box(\exists x\colon \boxminus H_A(z,x)) \rhd \Box(\exists y\colon \boxminus H_G(z,y)) \triangleq \Box[\circleddash(\exists x\colon \boxminus H_A(z,x)) \rightarrow (\exists y\colon H_G(z,y))]$$

Like in the case without hiding, the defining formula $\Box[\circleddash(\exists x\colon \boxminus H_A(z,x)) \rightarrow (\exists y\colon H_G(z,y))]$ (which is shorter than but equivalent to $\Box[\circleddash(\exists x\colon \boxminus H_A(z,x)) \rightarrow (\exists y\colon \boxminus H_G(z,y))]$, the original defining formula in [13]) says that $\exists y\colon \boxminus H_G(z,y)$ holds at least one step longer than $\exists x\colon \boxminus H_A(z,x)$ does. Below is a general rule for composing A-G specifications with safety assumptions and guarantees.

Theorem 3.2 *Assume that the tuples z, x, y, x_1, ..., x_n, y_1, ..., y_n of variables are pairwise disjoint. Then,*

1. *For* $1 \leq j \leq n$,
$$\models \Box\Big[(\exists x\colon \boxminus H_A(z,x)) \wedge (\exists y_1 \ldots y_n\colon \boxminus \bigwedge_{i=1}^{n} H_{G_i}(z,y_i)) \rightarrow (\exists x_j\colon H_{A_j}(z,x_j))\Big]$$
2. $$\models \Box\Big[\circleddash(\exists x\colon \boxminus H_A(z,x)) \wedge (\exists y_1 \ldots y_n\colon \boxminus \bigwedge_{i=1}^{n} H_{G_i}(z,y_i)) \rightarrow (\exists y\colon H_G(z,y))\Big]$$

$$\models \bigwedge_{i=1}^{n} \Box[\circleddash(\exists x_i\colon \boxminus H_{A_i}(z,x_i)) \rightarrow (\exists y_i\colon H_{G_i}(z,y_i))]$$
$$\rightarrow \Box[\circleddash(\exists x\colon \boxminus H_A(z,x)) \rightarrow (\exists y\colon H_G(z,y))]$$

The rule can be specialized for proving that an A-G specification refines another and that some module implements an A-G specification.

4 Example: A Token Ring

The preceding compositional framework can be applied in various contexts. We consider an example that illustrates the following common situation in component-based software development:

> A developer designs an application module that is to be combined with an existing module to form a complete system. It is known that the existing module complies with certain specification, but its code is not disclosed. The developer nevertheless needs to verify that the complete system is correct.

The example concerns proper interaction between a group of servers, arranged as a ring, and a group of clients, each connected to a distinct server. By circulating a unique token around

the ring, the servers module provides a mutual exclusion service to the clients module. When a client wants to enter the critical section, it sends a request to its server. The requested server, upon receiving the token from its predecessor server, will transmit the token (or any other equivalent representation) to the requesting client. Once the token is acquired, the client may proceed to the critical section. Upon exiting the critical section, the client sends the token back to its server, which will pass the token to the next server.

The safety property of the entire system, consisting of the servers and the clients modules, states that *at most one client is in the critical section at any time.* To apply the compositional framework, we envision that the servers module is specified by an A-G formula. We shall verify the safety property in a compositional way, given the A-G specification of the servers module and the code (system specification) of the clients module.

As illustrated in Fig. 1, $Server_i$ communicates with its client $Client_i$ via an input and an output channels: sin_i and $sout_i$. The input channel sin_i is for $Client_i$ to send a request or return the token to $Server_i$, while the output channel $sout_i$ is for $Server_i$ to send the token to $Client_i$. Both sin_i and $sout_i$ consist of three fields: sig, ack, and val of type boolean. A channel is clear and ready for sending when $sig = ack$. To send a message, the sender writes an appropriate value into val (*true* for a token and *false* for a request) and complements sig so that $sig = \neg ack$ to notify the receiver. The receiver reads the value of val and complements ack so that $sig = ack$ to clear the channel.

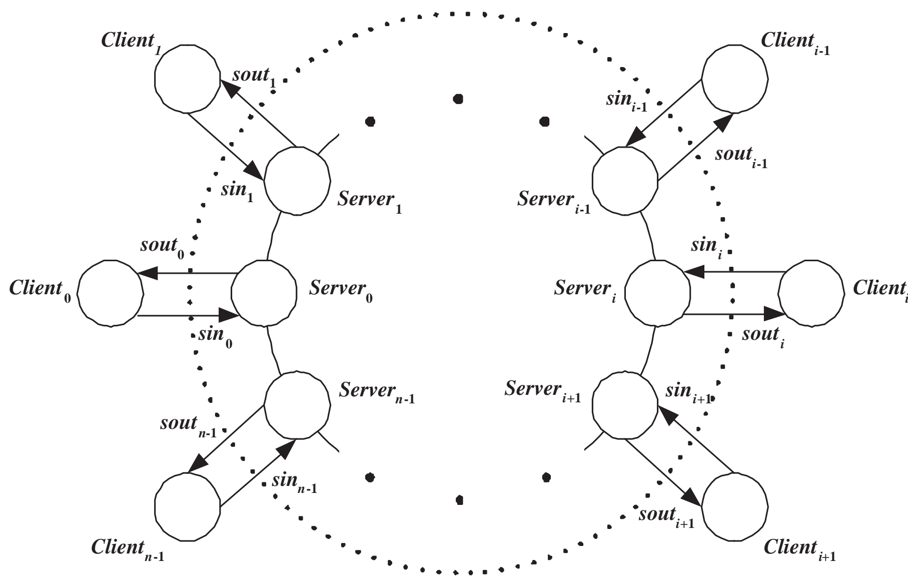

Figure 1: A token ring with n servers and clients. The dotted full circle cutting through sin and $sout$ indicates the boundary of the $Servers$ and the $Clients$ modules.

Let M_s denote the ring of servers $(\|_{i=0}^{n-1} Server_i)$ and M_c the collection of clients $(\|_{i=0}^{n-1} Client_i)$. The entire system is therefore the parallel composition of the two modules: $M_s \parallel M_c$. The two modules interact with each other via the channel variables $sin_0, \ldots, sin_{n-1}$ and $sout_0, \ldots, sout_{n-1}$. $Server_i$ controls the ack field of sin_i and the sig and val fields of $sout_i$, while $Client_i$

controls the sig and val fields of sin_i and the ack field of $sout_i$. Below are a few useful state and action formulas regarding a channel.

$$\begin{array}{lcl}
sinClear_i & \triangleq & sin_i.sig = sin_i.ack \\
sinToken_i & \triangleq & sin_i.sig \neq sin_i.ack \wedge sin_i.val = true \\
sinRequest_i & \triangleq & sin_i.sig \neq sin_i.ack \wedge sin_i.val = false \\
sinNotWritten_i & \triangleq & sin_i'.sig = sin_i.sig \wedge sin_i'.val = sin_i.val \\
sinNotRead_i & \triangleq & sin_i'.ack = sin_i.ack
\end{array}$$

Formulas $soutClear_i$, $soutToken_i$, $soutNotWritten_i$, and $soutNotRead_i$ are analogously defined, except that $soutToken_i$ is simplified as $sout_i.sig \neq sout_i.ack$.

4.1 A-G Specification of the Servers Module

What would be a reasonable specification for the servers module? For the servers module, the clients module is its environment. The servers module requires the cooperation of the clients module to provide the right service. Complying with the conventions of sending and receiving values through the channel variables, the servers' assumption about the clients should include the following:

1. Initially, no client has the token. To express this formally, we postulate an internal boolean variable $cliHasToken_i$ for each client i, whose truth value indicates whether client i is holding a token. Appropriate values for the channels incorporated, the initial condition should be:

   $$Init_{A_s} \triangleq \bigwedge_{i=0}^{n-1} \neg cliHasToken_i \wedge \neg sin_i.sig \wedge \neg sout_i.ack \wedge sin_i.val$$

   It is interesting to note that a single internal boolean variable for the entire clients module would not be adequate. The clients execute independently from one another, it is very well possible for two clients to hold a token at the same time if the servers behave incorrectly so as to generate an extra token in the system.

   The stuttering step for each client should be:

   $$\begin{array}{lcl}
   cliUnchanged_i & \triangleq & (cliHasToken_i' = cliHasToken_i) \wedge sinNotWritten_i \wedge \\
   & & soutNotRead_i
   \end{array}$$

   The stuttering step for the channel controlled by each server should be:

   $$serChanUnchanged_i \triangleq sinNotRead_i \wedge soutNotWritten_i$$

   The stuttering step for the channel controlled by the servers module should be:

   $$serChanUnchanged \triangleq \bigwedge_{i=0}^{n-1} serChanUnchanged_i$$

2. In each step, the clients are assumed to do one of the following:

   - Some client makes a request for the token by sending the request over the input channel of its server.

$$\begin{aligned} \mathit{cliMakeRequest} \triangleq\ & [\bigvee_{i=0}^{n-1} \neg \mathit{cliHasToken}_i \wedge \neg \mathit{cliHasToken}'_i \wedge \\ & \mathit{sinClear}_i \wedge \mathit{sinRequest}'_i \wedge \mathit{soutNotRead}_i \wedge \\ & (\bigwedge_{j \neq i} \mathit{cliUnchanged}_j)] \wedge \mathit{serChanUnchanged} \end{aligned}$$

   - Some client with a token sitting in its input channel, i.e., the output channel of its server, grabs and keeps the token.

$$\begin{aligned} \mathit{cliGrabToken} \triangleq\ & [\bigvee_{i=0}^{n-1} \neg \mathit{cliHasToken}_i \wedge \mathit{cliHasToken}'_i \wedge \\ & \mathit{soutToken}_i \wedge \mathit{soutClear}'_i \wedge \mathit{sinNotWritten}_i \wedge \\ & (\bigwedge_{j \neq i} \mathit{cliUnchanged}_j)] \wedge \mathit{serChanUnchanged} \end{aligned}$$

   - Some client holding a token sends the token back to its server, i.e., puts the token in the input channel of the server.

$$\begin{aligned} \mathit{cliReleaseToken} \triangleq\ & [\bigvee_{i=0}^{n-1} \mathit{cliHasToken}_i \wedge \neg \mathit{cliHasToken}'_i \wedge \\ & \mathit{sinClear}_i \wedge \mathit{sinToken}'_i \wedge \mathit{soutNotRead}_i \wedge \\ & (\bigwedge_{j \neq i} \mathit{cliUnchanged}_j)] \wedge \mathit{serChanUnchanged} \end{aligned}$$

   - Do nothing, or more precisely, do not change the values of the interface variables controlled by the clients or the postulated internal variables $\mathit{cliHasToken}_0, \ldots, \mathit{cliHasToken}_{n-1}$.

$$\mathit{cliUnchanged} \triangleq \bigwedge_{i=0}^{n-1} \mathit{cliUnchanged}_i$$

   The state transitions of the clients module should be:

$$\begin{aligned} \Box N_{A_s} \triangleq\ & \Box(\mathit{cliMakeRequest} \vee \mathit{cliGrabToken} \vee \mathit{cliReleaseToken} \vee \\ & \mathit{cliUnchanged}) \end{aligned}$$

The assumption of the servers module should be:

$$A_s \triangleq \mathit{Init}_{A_s} \wedge \Box N_{A_s}$$

The formula A_s specifies the assumed behavior of the clients module. It involves not only the clients' interaction with the servers at the channels, but also the internal state of a client regarding whether the client holds the token via the postulated variables $\mathit{cliHasToken}_0, \ldots, \mathit{cliHasToken}_{n-1}$. We omit the existential quantifications. We shall later try to relate $\mathit{cliHasToken}_0, \ldots, \mathit{cliHasToken}_{n-1}$ with the internal states of the clients when we compose the servers and the clients.

The guarantee of the servers module should state the following:

1. Initially, the servers module holds the token. Formally, we postulate an internal boolean variable $serHasToken_i$ for each server i, whose truth value indicates whether server i is holding a token. Appropriate values for the channels incorporated, the initial condition should be:

$$\begin{aligned} Init_{G_s} \triangleq \; & [\bigvee_{i=0}^{n-1} serHasToken_i \wedge (\bigwedge_{j \neq i} \neg serHasToken_j)] \wedge \\ & (\bigwedge_{i=0}^{n-1} \neg sin_i.ack \wedge \neg sout_i.sig) \end{aligned}$$

The stuttering step for each server should be:

$$\begin{aligned} serUnchanged_i \triangleq \; & (serHasToken_i' = serHasToken_i) \wedge sinNotRead_i \wedge \\ & soutNotWritten_i \end{aligned}$$

The stuttering step for the channel controlled by each client should be:

$$cliChanUnchanged_i \triangleq sinNotWritten_i \wedge soutNotRead_i$$

The stuttering step for the channel controlled by the clients module should be:

$$cliChanUnchanged \triangleq \bigwedge_{i=0}^{n-1} cliChanUnchanged_i$$

2. In each step, the servers module will do one of the following:

   - Pass the token to the next server.

$$\begin{aligned} serPassToken \triangleq \; & [\bigvee_{i=0}^{n-1} serHasToken_i \wedge \neg serHasToken_i' \wedge \\ & \neg serHasToken_{(i+1)\%n} \wedge serHasToken_{(i+1)\%n}' \wedge \\ & serChanUnchanged_i \wedge \\ & serChanUnchanged_{(i+1)\%n} \wedge \\ & (\bigwedge_{j=0}^{n-1} ((j \neq i) \wedge (j \neq ((i+1)\%n))) \rightarrow serUnchanged_j)] \end{aligned}$$

   - Grant the token to one of the clients with an outstanding request.

$$\begin{aligned} serGrantToken \triangleq \; & [\bigvee_{i=0}^{n-1} serHasToken_i \wedge \neg serHasToken_i' \wedge \\ & sinRequest_i \wedge sinClear_i' \wedge soutToken_i' \wedge \\ & (\bigwedge_{j \neq i} \neg serHasToken_j \wedge \\ & (sinClear_j \vee sinRequest_j) \wedge serUnchanged_j)] \wedge \\ & cliChanUnchanged \end{aligned}$$

   - Reclaim the token that is sitting in the output channel of some client, or the input channel of some server.

$$\begin{aligned} serReclaimToken \triangleq \; & [\bigvee_{i=0}^{n-1} \neg serHasToken_i \wedge serHasToken_i' \wedge \\ & sinToken_i \wedge sinClear_i' \wedge soutNotWritten_i \wedge \\ & (\bigwedge_{j \neq i} serUnchanged_j)] \wedge \\ & cliChanUnchanged \end{aligned}$$

- Do nothing, or more precisely, do not change the values of the interface variables controlled by the servers or the postulated internal variables $serHasToken_0$, ..., $serHasToken_{n-1}$.

$$serUnchanged \triangleq \bigwedge_{i=0}^{n-1} serUnchanged_i$$

The state transitions of the servers module should be:

$$\Box N_{G_s} \triangleq \Box(serPassToken \vee serGrantToken \vee serReclaimToken \vee serUnchanged)$$

The guarantee of the servers module G_s should be:

$$G_s \triangleq Init_{G_s} \wedge \Box N_{G_s}$$

Again, we omit the existential quantifications.

4.2 The Clients Module

The code of the clients module may be given as a system specification in LTL, shown in Fig. 2.

$$
\begin{array}{lcl}
Init_{M_c} & \triangleq & \bigwedge_{i=0}^{n-1} (c_state_i = 0) \wedge \neg sin_i.sig \wedge \neg sin_i.ack \wedge \\
 & & \quad sin_i.val \wedge \neg sout_i.sig \wedge \neg sout_i.ack \wedge \neg sout_i.val \\
Skip_c_i & \triangleq & (c_state'_i = c_state_i) \wedge (sin'_i = sin_i) \wedge (sout'_i = sout_i) \\
Act_c1_i & \triangleq & (c_state_i = 0) \wedge (c_state'_i = 1) \wedge (sin'_i = sin_i) \wedge (sout'_i = sout_i) \\
Act_c2_i & \triangleq & (c_state_i = 1) \wedge (sin_i.sig = sin_i.ack) \wedge sin_i.val \wedge (c_state'_i = 2) \wedge \\
 & & (sin'_i.sig = \neg sin_i.sig) \wedge \neg sin'_i.val \wedge (sin'_i.ack = sin_i.ack) \wedge \\
 & & (sout'_i = sout_i) \\
Act_c3_i & \triangleq & (c_state_i = 2) \wedge (sout_i.sig \neq sout_i.ack) \wedge (c_state'_i = 3) \\
 & & \wedge (sout'_i.ack = sout_i.ack) \wedge (sout'_i.sig = sout_i.sig) \wedge \\
 & & (sout'_i.val = sout_i.val) \wedge (sin'_i = sin_i) \\
Act_c4_i & \triangleq & (c_state_i = 3) \wedge (c_state'_i = 4) \wedge (sin'_i = sin_i) \wedge (sout'_i = sout_i) \\
Act_c5_i & \triangleq & (c_state_i = 4) \wedge (sin_i.sig = sin_i.ack) \wedge \neg sin_i.val \wedge (c_state'_i = 0) \wedge \\
 & & (sin'_i.sig = \neg sin_i.sig) \wedge sin'_i.val \wedge (sin'_i.ack = sin_i.ack) \wedge \\
 & & (sout'_i = sout_i) \\
Act_skip_c_i & \triangleq & (c_state'_i = c_state_i) \wedge (sin'_i.sig = sin_i.sig) \wedge \\
 & & (sin'_i.val = sin_i.val) \wedge (sout'_i.ack = sout_i.ack) \\
Act_c_i & \triangleq & Act_c1_i \vee Act_c2_i \vee Act_c3_i \vee Act_c4_i \vee Act_c5_i \vee Act_skip_c_i \\
N_{M_c} & \triangleq & \bigvee_{i=0}^{n-1} (Act_c_i \wedge (\bigwedge_{j \neq i} Skip_c_j)) \\
M_c & \triangleq & Init_{M_c} \wedge \Box N_{M_c}
\end{array}
$$

Figure 2: System specification of the clients module

5 Mechanization with a Proof Assistant

For the ease of mechanization, we have imposed in Section 3 a restriction that assumptions and guarantees be of the canonical form $\Box((\mathit{first} \rightarrow Init) \wedge N)$ $(\equiv Init \wedge \Box N)$, where N is an action

formula that relates the current state and the previous state of the system. There is one more obstacle.

Introducing existential quantification over a flexible variable of a temporal formula in general-purpose proof assistants is not as convenient as in hand proofs. The state of a system is typically represented as a tuple of named components, each corresponding to a flexible variable of the system. One is allowed to quantify over the entire tuple, but not its components individually. In particular, one may not universally quantify over some component while existentially quantify over another of the same tuple. It may be possible to divide the state of a system into the external and the internal parts represented as two separate tuples so that one can existentially quantify the internal part. However, as specifications are composed, it is common for an external variable to become internal in a different context.

We propose a way to avoid dividing the state of a system and yet be able to imitate the effect of hiding. The formulas in both premises of Theorem 3.2 are in a weaker form than the usual refinement relation between two canonical formulas with hidden variables, because the existential quantifications occur inside the $\Box$ operator. Like in the usual case, we may find appropriate state functions $f_1, \ldots, f_n, g$ that map from the free variables of $H_A, H_{G_1}, \ldots, H_{G_n}$, i.e., the tuples $z, x, y_1, \cdots, y_n$ of variables, to the domains of $x_1, \ldots, x_n, y$ and try to establish the validity of following formulas:

1. $\Box\Big[\boxminus H_A(z,x) \wedge \boxminus \bigwedge_{i=1}^{n} H_{G_i}(z,y_i) \rightarrow H_{A_j}(z, f_j/x_j)\Big]$, for $1 \leq j \leq n$.
2. $\Box\Big[\circleddash \boxminus H_A(z,x) \wedge \boxminus \bigwedge_{i=1}^{n} H_{G_i}(z,y_i) \rightarrow H_G(z, g/y)\Big]$.

The above formulas respectively are in the same form as those in the premises of the following rule, which is obtained from Theorem 3.2 by removing all existential quantifications.

$$\frac{\begin{array}{ll} 1. & \models \Box\Big[\boxminus H_A(z,x) \wedge \boxminus \bigwedge_{i=1}^{n} H_{G_i}(z,y_i) \rightarrow H_{A_j}(z,x_j)\Big], \text{ for } 1 \leq j \leq n \\ 2. & \models \Box\Big[\circleddash \boxminus H_A(z,x) \wedge \boxminus \bigwedge_{i=1}^{n} H_{G_i}(z,y_i) \rightarrow H_G(z,y)\Big] \end{array}}{\models \bigwedge_{i=1}^{n} \Box[\circleddash \boxminus H_{A_i}(z,x_i) \rightarrow H_{G_i}(z,y_i)] \;\rightarrow\; \Box[\circleddash \boxminus H_A(z,x) \rightarrow H_G(z,y)]}$$

This rule is in fact identical to Theorem 3.1, except that the variables are made explicit and tuples of variables with different names are assumed to be disjoint. Our idea is to use Theorem 3.1 to bring out the proof obligations. The internal variables are declared as state functions separate from the tuple that represents the state of the system and the needed refinement mappings are introduced as axioms that relate those functions with appropriate free variables. Specifically, "$x_i = f_i$" is asserted for some f_i that is a state expression involving only free variables of $H_A, H_{G_1}, \ldots, H_{G_n}$, i.e., $z, x, y_1, \cdots, y_n$, and so on. Such refinement mappings should be easily checked by hand or by some external means for validity. As long as one follows the principles in defining a refinement mapping, there should be no inconsistency resulted from introducing the axioms.

6 Continuation of the Token-Ring Example

In PVS, the internal variables $serHasToken_0, \ldots, serHasToken_{n-1}$, $cliHasToken_0, \ldots, cliHasToken_{n-1}$ are declared separately as two functions as follows:

```
serHasToken: [pid -> [state -> bool]]
cliHasToken: [pid -> [state -> bool]]
```

Given the A-G specification $A_s \rhd G_s$ of the servers module and the system specification (the code) M_c of the clients module, we apply the composition rule to prove the desired mutual exclusion property of the system as follows:

$$G \triangleq \Box(\bigwedge_{i \neq j} \neg atCrit_i \vee \neg atCrit_j)$$
$$\Phi_{mutex} \triangleq (A_s \rhd G_s) \wedge (true \rhd M_c) \rightarrow (true \rhd G)$$

where $atCrit_i = (c_state_i = 3)$ and $atCrit_j = (c_state_j = 3)$. The required mapping is:

$$cliHasToken_i = ((c_state_i = 3) \vee (c_state_i = 4))$$

7 Discussion

Assuming a fixed number of nodes and with a suitable encoding of finite-domain variables with booleans, the token-ring example can entirely be dealt with using QPTL which is a subset of LTL where the type of every variable is boolean. The advantage of using QPTL is that QPTL is a decidable language and the verification task can *in principle* be carried out by a QPTL model checker. Unfortunately, no existing model checkers support QPTL. One get-around is to remove the existential quantifiers by encoding the needed refinement mapping as part of the client module, which has to be done by hand.

References

[1] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for infinite games on recursive graphs. In *Proceedings of the 15th International Conference on Computer-Aided Verification, LNCS 2725*, pages 67–79, 2003.

[2] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for recursive game graphs. In *Proceedings of TACAS, LNCS 2619*, pages 363–378, 2003.

[3] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[4] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdziński, and F.Y.C. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer-Aided Verification, LNCS 2404*, pages 428–441. Springer-Verlag, 2002.

[5] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In *Proceedings of the 14th International Conference on Computer-Aided Verification, LNCS 2404*, pages 414–427. Springer-Verlag, 2002.

[6] 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.

[7] L. de Alfaro. Game models for open systems. In N. Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday, LNCS 2772*, pages 269–289. Springer, 2003.

[8] L. de Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM Symposium on Foundations of Software Engineering*, pages 109–120, 2001.

[9] C.A.R. Hoare. An axiomatic basis for computer programs. *Communications of the ACM*, 12(8):576–580, 1969.

[10] C.B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.

[11] B. Jonsson. A model and proof system for asynchronous networks. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 49–58, 1985.

[12] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):1–42, March 1994.

[13] 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.

[14] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[15] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[16] N.A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

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

[18] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[19] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[20] J.-W. Teng and Y.-K. Tsay. Composing temporal-logic specifications with machine assistance. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings of the 12th International Formal Methods Europe Symposium (FME 2003), LNCS 2805*, pages 719–738. Springer, September 2003.

[21] 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.