A FORMAL APPROACH TO RELIABLIE NETWORK SOFTWARE

Shoji Yuen, Keigo Imai, Ryo Suetsugu, Kiyoshi Agusa

Graduate School of Information Science,Nagoya University, Japan

ABSTRACT

We have been investigating *communicating processes*, where the primitive computation is synchronous communication between processes. A process is a "black box" to be identified only by observing communications with the environment. Various behavioral characterizations of communicating processes have been studied over the last couple of decades. The model is good for realizing concurrent behavior of network software. We realize the behavior of concurrent software based on equivalences as the fundamental semantics of programming languages with concurrent features. During the period of the COE project, we have investigated the following topics with the aim of improving the reliability of network software. (1) A meta algebraic framework defined by SOS(Structural Operational Semantics) for timed process calculi; (2) Web application modeling; (3) user interface behavioral modeling; (4) timed extension of a mobile calculus; (5) a network framework for functional programming language Haskell; and (6) behavioral modeling for embedded systems.

1. INTRODUCTION

The importance of network computing has been increasing in accordance with the rapid development of computer network. In addition to the reliability of computer network itself, the reliability of software working over computer network becomes a critical issue. The distinguishing feature of network software is that many programs run concurrently interacting over network. Generally, each program runs asynchronously without any global synchronous mechanism. Network software may run efficiently on many processing units and the separation of component-wise functionality provides a better prospects in software design. On the other hand, network software is inherently difficult to handle due to the following properties: (1) Its behavior is inherently nondeterministic. The state of an entire software system is difficult to determin due to the lack of a global control mechanism. (2) Interaction changes the behavior of each component from outside of the component in the course of computation. Consequently, it is not possible to model the computation by a function identified by an input/output relation. These features are well recognized in modeling network software, and various kinds of extensions to alleviate those difficulties have been proposed for decades.

We have been focusing on a particular model of "communicating processes" due to its simplicity and semantic clarity. The basic semantics is usually formalized as equivalences based on bisimulation relations. An equivalence is defined to form an algebra with respect to the operators that construct terms; that is, the equivalence is a congruence relation with respect to the operators. This algebraic characterization is often called "a process algebra" being a major realization of communicating processes. Hoare's CSP [1] is known to have an algebraic semantics when processes are defined to be equivalent if they have the same deadlock capability. Milner's CCS [2] has an algebraic semantics when communicating capability of processes is identical at every point of communication. ACP (Algebra of Communicating Processes) by Bergstra and Baeten, as well as by many other Dutch researchers [3] first defines the equivalences along with the operational semantics that fits to those equivalences. Beginning with these pioneering work, various formal systems have been proposed based on the similar approaches. We believe this algebraic characterization assures the better treatment of concurrent behavior of network software.

Based on the communicating process model, we are attempting to realize network concurrent software with the aim of improving the behavioral reliability by providing verification techniques based on the calculi. We have been concentrating on modeling the existing software with concurrent features and on extending the calculi for the modeling technique.

First, we present the extensions for timed behavior. Process calculi deal with "temporal" properties of communications. However, in network software, it is also important "when" communications happen. One typical important mechanism is "timeout", in fact, "timeout" is found to be the primitive action for all timed behavior, the length of time until the behavior change is a critical issue. An inappropriate length of time may crash the entire software. We investigated an extension introducing "time-tick" to process calculi, first to conventional process calculi, and then to the π-calculus.

Next, we show an application to a more practical system of Web applications. A Web application is a server-side process that reacts to requests from clients. By extending the actions, we model the behavior of Web applications. Here, an action is extended to hold equations (or inequations) to alter the behavior due to values exchanged with clients.

For more direct modeling of network software, we studied a programming language based on the π-calculus as the joint research with the NTT Communication Laboratory. The language "Nepi" is a direct implementation of π calculus in Common Lisp. Since built-in Common Lisp functions can be used as communications in "Nepi", practical network programs are simply described based on communicating processes. For the formal treatment of "Nepi" behavior, we illustrate a fundamental framework for "user interface", which conventional software finds difficult to handle.

Finally we present a summary of work in progress conducted by research associates of the COE project in our research group.

This report consists of brief summaries of work conducted during the COE project. Section2 describes a timed extension of the ordered SOS format, and we discuss the meta framework for process calculi with discrete timing. Section3 describes another timed extension for π calculus. Section4 explains modeling web applications based on interactive state transition systems, called "web automaton". Section5 presents the programming language "nepi". Section6 outlines work in progress and Section 7 offers concluding remarks.

2. TIMED EXTENSION FOR THE ORDERED SOS FORMAT

(This is a joint research project with Dr. Irek Ulidowski (University of Leicester)[4].)

Operational Definition by SOS rules

We extended the ordered SOS format for process calculi with relative discrete time. As stated in the Introduction, many process calculi have been proposed. Although each calculus has its own purpose, generally the algebraic characterization with congruence relations is one of major goals of the calculi. As a proof technique, there are many similarities among the calculi. The operational semantics of a process calculus is defined by *structural operational semantics*, SOS for short, originally proposed by Plotkin[5]. A transition $P \xrightarrow{\alpha} P'$, meaning P becomes P' by communication α, holds if and only if the transition is inferred by the SOS rules. Figure 1 shows an example of SOS rules for CCS without recursion. The rules compositionally define the transitions of the operators, prefix $\alpha.(\_)$, choice $\_ + \_$, parallel composition $\_|\_$ and restriction $\_\backslash A$.

$$\frac{}{\alpha.X \xrightarrow{\alpha} X}\,r_\alpha$$

$$\frac{X \xrightarrow{\alpha} X'}{X + Y \xrightarrow{\alpha} X'}\,r_{+l} \qquad \frac{Y \xrightarrow{\alpha} Y'}{X + Y \xrightarrow{\alpha} Y'}\,r_{+r}$$

$$\frac{X \xrightarrow{\alpha} X'}{X|Y \xrightarrow{\alpha} X'|Y}\,r_{|l} \qquad \frac{Y \xrightarrow{\alpha} Y'}{X|Y \xrightarrow{\alpha} X|Y'}\,r_{|r}$$

$$\frac{X \xrightarrow{\lambda} X', Y \xrightarrow{\bar{\lambda}} Y'}{X|Y \xrightarrow{\tau} X'|Y'}\,r_c \qquad \frac{X \xrightarrow{\alpha} X'}{X\backslash A \xrightarrow{\alpha} X'\backslash A}\,(\alpha \notin A)\,r_\backslash$$

Fig. 1. SOS rules for CCS

Ordered SOS

More general formats of rules such as the General SOS format (GSOS for short), the tyft/tyxt format[6] and the ntyft/ntyxt[7] format, are known to define a bisimulation equivalence as congruences. In the GSOS format, each rule has the following form.

$$\frac{\{X_i \xrightarrow{a_{ij}} Y_{ij}\}_{i\in I, j\in J_i} \cup \{X_i \overset{b_{ij}}{\not\rightarrow}\}}{f(\vec{X}) \xrightarrow{c} C[\vec{X},\vec{Y}]}$$

where X_i and Y_j are variables, $\vec{X}$ and $\vec{Y}$ are tuples of variables and $C[\vec{X},\vec{Y}]$ is a term containing variables up to $\vec{X},\vec{Y}$. The GSOS format is restrictive in that no term structure in the premises and no multi-level term on LHS of the transition in the conclusion.

If all rules are in the GSOS format, the strong bisimulation equivalence and the ready simulation equivalence are congruent with respect to the defined operators[8]. GSOS allows negative transition in the premises of SOS rules, as $X \overset{\alpha}{\not\rightarrow}$, which holds if no communication α is possible from X. Ordered SOS format proposed by Ulidowski and Phillips is equivalent to GSOS, but OSOS has a advantage that no negative premises are permitted. This limitation allows simpler treatment in the proof, especially for weak semantics where unobservable τ action is abstracted away. Instead of negative premises, an OSOS format is accompanied by an order relation between SOS rules. A higher ordered rule has priority over lower ordered ones. Suppose following r_a and r_b are related as $r_a > r_b$.

$$\frac{X \xrightarrow{a} X'}{f(X) \xrightarrow{a} t}\,r_a \qquad \frac{X \xrightarrow{b} X'}{f(X) \xrightarrow{b} u}\,r_b$$

Then, r_b is applicable only if r_a is not applicable, meaning that r_b effectively has a negative premise $X \overset{a}{\not\rightarrow}$. We write $higher(r)$ for the set of rules ordered higher than r, namely, $\{r'|r < r'\}$.

A rule of the OSOS format is in the same form as one of GSOS only with positive premises. For rule r, we write $actions(r, i)$ for the label of the i-th transition in the premises and $active(r)$ for the index set of variables appearing on the LHS of transitions in premises. rules(f) denotes the set of rules where the operator appearing in LHS of conclusion is f.

The eager bisimulation equivalence and branching bisimulation preorder[1] is congruent with respect to all defined operators except choice like operators if the order satisfies the

[1] In weak semantics, we need to consider the divergence of infinite τ se-

certain set of conditions[9]. The eager bisimularity is slightly stronger than the conventional observation equivalence[2]. Therefore, if a pair of processes is eager bisimular, then the processes are also observationally equivalent.

To attain a well structured weak semantics, a τ transition must be kept unobservable and independent. In this respect, if there exists τ transitions, it must be in the following form:

$$\frac{X_i \xrightarrow{\tau} X_i'}{f(X_1,\cdots,X_i,\cdots,X_n) \xrightarrow{\tau} f(X_1,\cdots,X_i',\cdots,X_n)}\ \tau_i$$

A rule of this form is called a τ-rule written as τ_i. Another particular form is called a *choice rule* as follows:

$$\frac{X_i \xrightarrow{\alpha} X_i'}{f(X_1,\cdots,X_i,\cdots,X_n) \xrightarrow{\alpha} X_i'}\ \tau^i$$

When α is τ, it is written as τ^i. We write $tau(i)$ for either τ_i or τ^i. If any τ^i rule exists for f, then f must be a choice operator like '+' in CCS.

These two forms of rules are found to be a key to giving a well structured operational semantics. These two forms are supposed to be disjoint in arguments of defining operators. For this purpose, we have the following two types of operators, τ-preserving and τ-sensitive.

$$\textbf{if } \tau \in actions(r,i) \textbf{ then } r = tau(i) \qquad (1)$$

$$\textbf{if } i \in active(rules(f)) \textbf{ then } tau(i) \in rules(f) \textbf{ and } (\textbf{either } tau(i) = \tau_i \textbf{ or } tau(i) = \tau^i) \qquad (2)$$

$$\textbf{if } i \in active(rules(f)) \textbf{ then } tau(i) = \tau_i \qquad (3)$$

$$\textbf{if } i \in active(r)\ tau(i) = \tau^i \textbf{ then } r \text{ is a choice rule} \qquad (4)$$

$$\textbf{not}(tau(i) < tau(i)) \qquad (5)$$

$$\textbf{if } r' < r \textbf{ and } i \in active(r) \textbf{ then } r' < tau(i) \qquad (6)$$

$$\textbf{if } tau(i) < r \textbf{ and } i \in active(r) \cup active(higher(r')) \textbf{ then } r' < r \qquad (7)$$

$$\textbf{if } i \in \text{implicit-copies}(r) \textbf{ then } r < tau(i) \qquad (8)$$

Fig. 2. Conditions for eager bisimulation preorder

In figure 2, implicit-copies(r) is the index set of variables appearing both sides of the transition in the conclusion of r.

Let f be an operator with $\bigcup_{r\in \text{rules}(f)} active(r) \neq \emptyset$. The operator f is *τ-preserving* if the set of rules and the ordering on the rules satisfy (1)–(8). The operator f is *τ-sensitive* if it has a silent choice rule for one of its active arguments and the set of its rules and the ordering on the rules satisfy (1)–(2) and (4)–(8).

quence. $P \sqsubseteq Q$ means P and Q are equivalent but P may diverge if Q may diverge.

[2]The eager bisimularity does not allow τ transitions after observable transition.

The "rooted" eager bisimulation preorder distinguishes τ-transition at the root of operators with choice rules. Strengthening an equivalence to a rooted version of the equivalence

The following is an informal presentation of the main characteristics of process languages. For a more precise presentation, please refer to [4].

Theorem 1 *[10, 4] If operators defined in a process language are partitioned into τ-preserving, τ-sensitive, and operators with no active arguments, then all operators preserve the rooted eager bisimulation preorder.*

Timed Extension of Ordered SOS

We add the notion of discrete relative time by introducing the "time-tick" transition. Here, $P \xrightarrow{\sigma} P'$ means that P becomes P' by letting time pass for one tick, where σ is a special label for one unit of time. The time tick is different from the other labeled transitions in nature. We have been investigating what class of SOS format is appropriate for the process languages. For example, the following rules violate the notion of "time".

$$\frac{X \xrightarrow{\sigma} Y}{f(X) \xrightarrow{a} Y} \qquad \frac{X_1 \xrightarrow{\sigma} Y_1, X_2 \xrightarrow{a} Y_2}{f(X_1,X_2) \xrightarrow{\sigma} t} \qquad (9)$$

Both rules are legitimate SOS rules if σ is considered as an action. However, these rules clearly oppose the notion of time passage since the first rule "eliminates" time passage whereas the second rule "creates" it. This motivates further investigation into how timed behavior should be modeled in SOS formats. Therefore, a time tick transition $\xrightarrow{\sigma}$ must be derived by time tick transitions (possibly none) of its arguments.

Our approach to obtaining an appropriate timed process language is to extend the untimed process languages without breaking the algebraic property and with the natural time property at the same time. For example, CCS '+' is usually extended with a timed rule:

$$\frac{X \xrightarrow{\sigma} X', Y \xrightarrow{\sigma} Y'}{X + Y \xrightarrow{\sigma} X' + Y'}$$

The time tick is usually designed not to resolve the choice. Such operators have the following type of rules.

$$\frac{\{X_i \xrightarrow{\sigma} X_i'\}_{i\in I}}{f(\vec{X}) \xrightarrow{\sigma} f(\vec{X'})}$$

where I is the set of active arguments and $X_j' = X_j$ for $j \notin I$. This type of operators is called a 'time-preserving' one. The other type of operator is called a 'time-altering' one. For example, the time-out operator proposed by Hennessy and Regan is time-altering.

$$\frac{X \xrightarrow{a} X'}{\lfloor X \rfloor (Y) \xrightarrow{a} X'}\, r_a \qquad \frac{X \xrightarrow{\tau} X'}{\lfloor X \rfloor (Y) \xrightarrow{\tau} X'}\, r_\tau \qquad \frac{}{\lfloor X \rfloor (Y) \xrightarrow{\sigma} Y}\, r_\sigma$$

with the order of $r_\sigma < r_\tau$. A time-altering operator changes the structure of terms. By placing certain conditions on OSOS formats, we can guarantee the basic properties of time passage.

Results for Timed Extensions

In the extensions, not only the σ-rules and the order between them but also the order between σ-rules and τ-rules are added according to the 'strength' of the arguments. In the time-preserving extension of the CCS '+' operator, following three σ-rules and order between them.

$$\frac{X \xrightarrow{\sigma} X'}{X + Y \xrightarrow{\sigma} X'}\, r_{\sigma 1} \qquad \frac{Y \xrightarrow{\sigma} Y'}{X + Y \xrightarrow{\sigma} Y'}\, r_{\sigma 2}$$

$$\frac{X \xrightarrow{\sigma} X', Y \xrightarrow{\sigma} Y'}{X + Y \xrightarrow{\sigma} X' + Y'}\, r_{\sigma 12}$$

with the order of $r_{\sigma 1} < r_{\sigma 12}$ and $r_{\sigma 2} < r_{\sigma 12}$. Under the maximal synchrony assumption[11], neither $r_{\sigma 1}$ nor $r_{\sigma 2}$ becomes enabled, but we systematically add σ-rules to the active arguments.

For the time-out operator, r_σ has no premise although the first argument is active. To maintain the structure with respect to the weak semantics, generally the τ-transition has to be tried first. r_τ should be tried first prior to r_σ. Actually, $\{r_a, r_\tau, r_\sigma\}$ with $r_\tau > r_\sigma$ is an time altering extension of $\{r_a, r_\tau\}$ with no order.

A detailed explanation of the conditions for the time preserving extension and the time altering extension requires extra technical definitions and some more auxiliary notions. The precise definitions appear in the literature[4].

For a rebo process language with time passage, if the set of operators is partitioned into time-preserving and time-altering[3], then it is called a timed rebo language.

Here we just present the basic results for timed rebo process languages.

Theorem 2 *[4] A timed* rebo *process language preserves timed rooted eager bisimulation preorder.*

The basic timed property is the time determinacy property: If $p \xrightarrow{\sigma} p'$ and $p \xrightarrow{\sigma} p''$, then $p' \equiv p''$. The time determinacy property is important because the time passage is deterministic in nature.

Theorem 3 *[4] A timed* rebo *process language has the time determinacy property.*

[3]We need some more technical conditions in the precise presentation.

We also investigated several other timed properties, such as the maximal process property: if $p \xrightarrow{\tau} p'$ then $p \not\xrightarrow{\sigma}$. In the OSOS format, we simply order σ-rules under τ generating rules. For example, we order σ-rules of the CCS composition under σ-rules.

$$\frac{X \xrightarrow{\lambda} X', Y \xrightarrow{\bar{\lambda}} Y'}{X|Y \xrightarrow{\tau} X'|Y'}\, r_{|12} \qquad \frac{X \xrightarrow{\sigma} X', Y \xrightarrow{\sigma} Y'}{X|Y \xrightarrow{\sigma} X'|Y'}\, r_{\sigma 12}$$

with the order of $r_{\sigma 12} < r_{|12}$.

3. A TIMED EXTENSION OF A MOBILE CALCULUS

(This is joint work undertaken with Mr. H. Kuwabara (Nagoya University)[4][12].)

We propose equivalences and preorders with congruence properties for a timed extension of the π-calculus. Furthermore, we present a timed extension of syntax and basic operational semantics to this calculus. The derived timed bisimulation relations are shown to be non-input congruent. These timed bisimularities equalize the bisimular processes not only in actions but also in the timing of the actions. In order to model hard deadlines, we propose a more relaxed bisimulation, called the *delay time order* which relates a process behaving 'faster' in action to one with the same communication capability. We show that the delay time orders are non-input congruent where the 'time-insensitive' composition is allowed.

In our timed extension, the time-out operation is modeled by the choice with the delay prefix. The non-input congruence property ensures that adding the time-out operation does not break the delay time order provided that the length of time for the time-out is fixed. In this respect, the weak delay time order relation is a useful foundation for proving the correctness in practical use. This relation is congruent for contexts in which composition is 'time-insensitive' in the sense that time-passing actions do not change the status of composing processes. This restriction is not generally a large obstacle in the server/client systems because a server is generally time-insensitive since it process the requests from clients at any time. The detailed presentation of the example is in [12]

4. WEB APPLICATION MODELING

(This is the joint project with Mr. Keishi Kato(Hitachi Software Engineering)[13]).

We have proposed a behavioral model of Web applications, called 'Web Automata', based on the MVC(Model View and Control) model architecture. The MVC model architecture separates design concerns to improve overall software

[4]This research is mostly done while Mr. Kuwabara was a research associate of the COE project

quality. Since the architecture defines the abstract outline of the configurations, there is a broad gap between coding Web applications and their behavioral property.

The behavior of a Web application with dynamic contents is modeled as an extension of links-automata proposed by Stotts et.al. with the constraint-logic feature of the Extended Finite Automata (EFA) by Sarna-Starosa and Ramakrishna. As extended in the model checking techniques, we view a Web application as a data-independent system, where variables appearing in link parameters and form inputs are attached to each page.

We present a testing framework for Web applications based on the behavioral model, showing that it provides testing criteria for Web applications when we focus on the loops of the automata. Our framework is applied to the Jakarta Struts by presenting the extended configuration schema of Struts in order to describe the Web automata directly.

We focused on the property of Web applications that revisiting same pages often results in the similar transitions of pages. (Figure 3) Testing criteria are presented based on the number of revisits to pages, thus the more a testing sequence revisits same pages, the more refined test it is. Although the criterion depends on the property of the Web application under the test, in many cases we expect that the simplest criterion covers the interesting test sequences where the number of revisits is at most one.

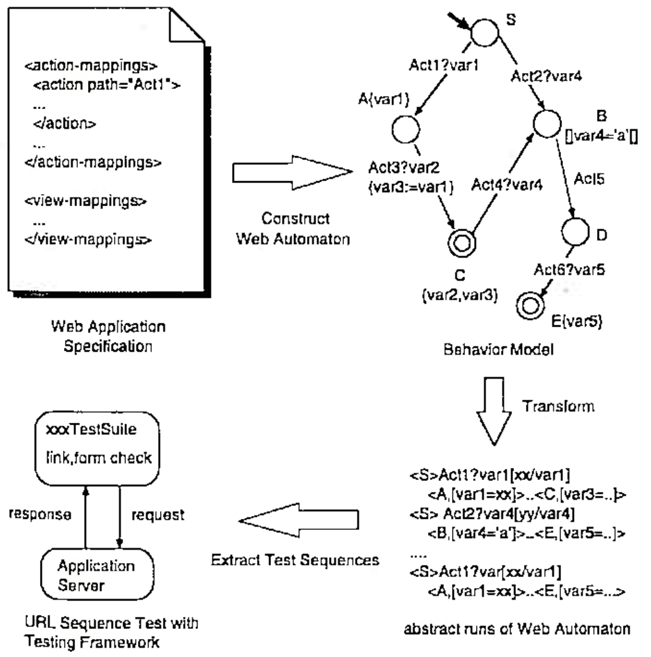


Fig. 3. Testing Frame in Web Automata

The following is an example of a simple library system modeled by the Web automaton. The library system features a simple management functionality such as searching for books for users as well as updating book information and logging the borrowing and return of books for managers with logging into the management account. (Figure 4)

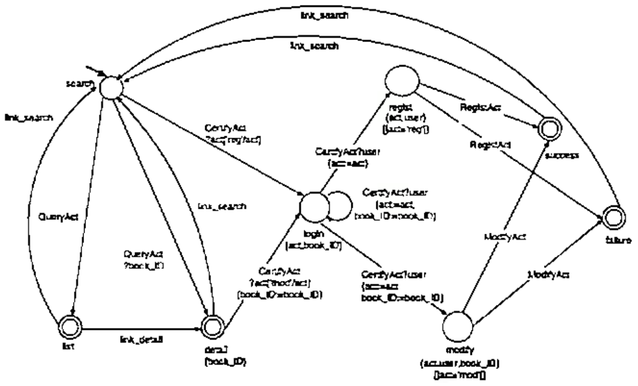


Fig. 4. Simple Library Management System

The web automaton model for this example is given as an EFA Each transition may have equations as constraints, and if there is an instance in which the equations are satisfied, the transition may occur. Generally there are infinitely many instances for a transition due to value assignment to variables. If they make no difference to the state transitions, those instances may be regarded as equivalent. The traces of real instances are called *concrete runs*, while those traces with uninstantiated variables are called *abstract runs*. Borrowing from this EFA technique , we show that the behavior of a Web application is recognized as a finite set of abstract runs.

We implemented an experimental prototype for a testing sequence generator following abstract runs, counting the number of testing sequences according to our criterion. From a theoretical point of view, the number of sequences increases exponentially, which implies that the general complete test becomes unrealistic. However, if n in criterion T^n is small, where T^n is the set of tests that visit the same pages n times at most, a complete test is still possible.

Criteria	T^0	T^1	T^2
Number of Test Sequences	9	132	1947

5. USER INTERFACE BASED ON COMMUNICATING PROCESSES

(This is the joint work with Mr.K.Mano(NTT), Dr.Y.Kawabe(NTT) and Mr.H.Kuwabara(Nagoya Univ.)[14])

A name-passing style Graphic User Interface (GUI) programming is proposed in the programming language Nepi, the operational semantics of which is based on the rendezvous-style name-passing communication of the π-calculus. Nepi is able to express timed behavior by combining the wait prefix with the external choice. We model GUI programs by using channel-based behavioral characterization. We propose a pair of extended syntax elements '?g' and '!g' in Nepi to generate and terminate graphic components. The graphic components are accompanied by event handling processes that convert a specified instance of name-passing event. In the extended version of Nepi, a GUI program is described as a composition of graphic components, event handling processes, and function processes that implement a real function. We present an implementation of a GUI extension for the Nepi programming language on Allegro Common Lisp to illustrate the features of name-passing style GUI programming in Nepi with examples.

GUI modeling by name passing

We model a GUI component regarded as comprising the following three types of communicating processes.

1. **Graphic component:** A process for representing the appearance of a GUI component, such as buttons. It has appropriate properties such as size or color. Each graphic component knows the names of relevant event-handling processes. The environment manages the creation and termination of a graphic component by passing names.
2. **Action process:** A process that performs the actual task. It corresponds to the call-back function in usual GUI programming. Upon receiving the appropriate name from the associated event-handing process, it actually reacts to the input and communicates with the relevant processes.
3. **Event-handling process:** A process that detects the occurrence of a raw event and passes on notification of the occurrence through a name created in advance. A graphic component is assumed to create an event-handling process for each event when a request is sent by an action process.

Figure 5 shows an overview of the behavioral model. In addition to the three types of process mentioned above, it is assumed that there is one environment process that controls the graphic environment. The environment process receives a request from an action process to initiate graphic components. Then, the graphic component creates the event-handling processes requested by the action process. The graphic component informs the created event handling processes of the action process by passing on the name of the relevant action process. In this respect, a typical name passing facility is used. After this initial procedure is complete, every time an event is accepted the event-handling process in charge sends this information to the action process followed by some reaction from the graphic component.

After creating a graphic component event-handling processes are produced according to the properties of the graphic component. The event-handling processes interface with the raw events from the real environment such as button clicks

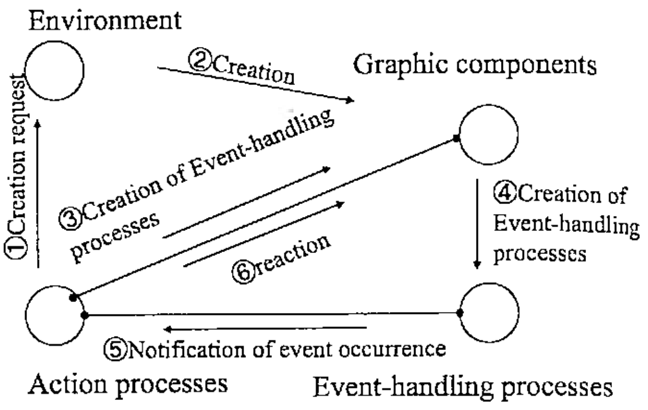
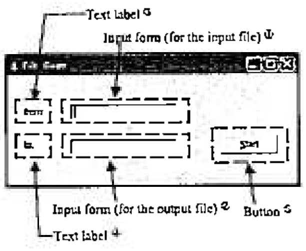

Environment
②Creation
Graphic components
①Creation request
③Creation of Event-handling processes
④Creation of Event-handling processes
⑥reaction
⑤Notification of event occurrence
Action processes
Event-handling processes

Fig. 5. Behavioral Model for GUI

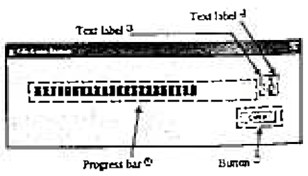
on the corresponding names. When closing a graphic component, a privileged name has to be sent to the component to terminate all sub-components.

Example

We practiced on a programming example of a file copier that copies one file to another showing the progress bar. First, the program displays the main window (figure 6(a)) on the screen. This window consists of a start button and two input forms. When the start button is pressed, the program creates a progress-bar window (figure 6(b)), and starts the procedure of copying the file. This procedure terminates only when (i) the file transfer is completed; (ii) a user clicks the cancel button on the progress-bar window; or (iii) the start button is untouched for 60 seconds.


Text label ③
Input form (for the input file) ①
Start
Input form (for the output file) ②
Button ⑤
Text label ④

(a) Window Input.



(b) Progress Bar.

Fig. 6. Example of a file copier.

All actions and events are handled by name-passing according to the behavioral modeling presented above.

Towards a Formal Treatment of GUI

As for the formal treatment of GUI operations, we have investigated a reduction semantics for GUIs. Using the model checking technique instead of conventional testing to verify the behavior, we expect to design a more reliable GUI. A detailed definition and discussion are presented in [14].

6. WORK IN PROGRESS

6.1. Functional Programming for Communicating Processes

This research proposes a network programming framework for Haskell, called PiMonad. Programming language Haskell is one of the major functional programming languages in use, gaining popularity with the features of non-strictness and strict typing. Together with the monad extension mechanism, Haskell features excellent flexibility for non-functional description such as for the loop structure, or for the I/O interface.

The aim of this research is to provide a network programming framework for analysis and reasoning about programs written in Haskell extended with a communication feature. The PiMonad framework is a lightweight implementation of asynchronous localized π-calculus[15]. The key point is the encoding of asynchrony and locality, which makes our implementation simple.

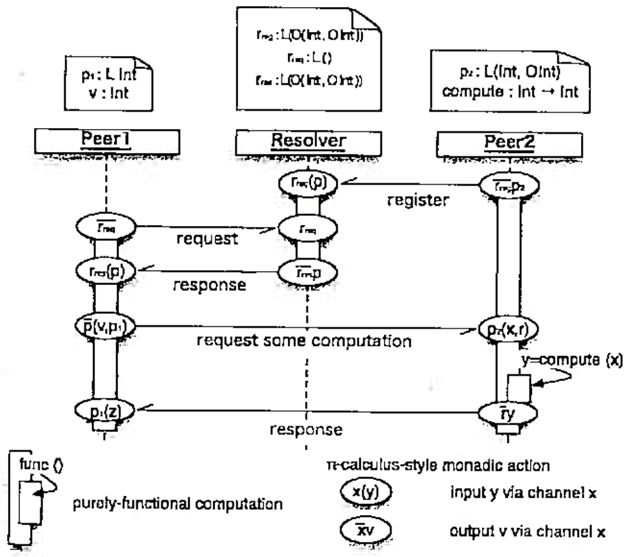


Fig. 7. Communication with types in PiMonad

We have developed a tracing tool in this framework that follows one path according to the asynchronous local π calculus. Since we directly describe the system based on the calculus, the behavior is assured to follow the abstract operational semantics. Based on this infrastructure, we plan to develop a verification tool as future work. Furthermore, we intend to devise a systematic method to verify temporal properties such as the deadlock free property.

6.2. Modeling Behavior of Embedded Systems

We have applied the communicating process model to the behavioral analysis of AIBO robot software as an example of small scale embedded software. AIBO is an entertainment robot dog developed by SONY Corporation. One of AIBO's outstanding features is that the robot is fully programmable in C++. Because the AIBO OS and the OPEN-R environment offer full programming capability, AIBO is an excellent platform for realizing small scale software for embedded systems.

We modeled AIBO software by the π-calculus[15] to observe the message passing flow between components according to the source code structure. In AIBO programming, concurrent object behavior is controlled by wait and notify signals. From our programming experience, the mistreatment of these signals is the major source of unexpected behavior of AIBO programs. Tracking these signals, we approximate the flow of control to detect deadlock that are not obvious from source programs in cases where the AIBO stopped working unintentionally.

Conceptually, objects run concurrently connected by one-to-one links. Communication over these links occurs asynchronously. The basic behavior of asynchronous message passing is as follows. (1) An object notifies other objects that a link is ready according to the link connections. (2) An object on the other side sends a message with a notify signal. Since all communications are asynchronous, the execution is never blocked except when a link waiting to receive a message. The scheduler controls which object runs next. Therefore, modeling the scheduler and tracking ready-notify signals, it is basically sufficient for capturing the abstract execution of AIBO software.

Based on this idea, we abstractly translate an AIBO program into a π calculus expression by the following steps.

Step 1 C++ source codes are sliced to extract message passing behavior;

Step 2 Sliced codes are translated to π processes along the OPEN-R object structure

Step 3 The scheduler is composed from the connection information between ports; and

Step 4 The system is described as the composition of all object processes and the scheduler.

The description obtained by the translation is supposed to track all the behavior of the AIBO program. By analyzing the description with model checking tools, we are able to verify the behavioral properties of the communicating objects

It is commonly seen in model checking practices that the behavioral composition generates a number of states leading to the intractable state explosion. Figure 8 shows decomposition verification for the deadlock free property by finding disjoint groups with respect to communications. The groups are

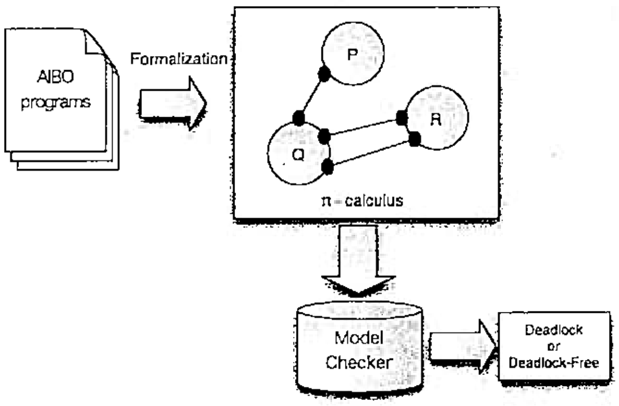


Fig. 8. Deadlock Free Property Analysis

found by checking the communication ports in the π process description. Using this technique, we have shown an example of 484 LOC partitioned into two groups. The deadlock free property was checked in 442 seconds and 104 seconds for the respective groups, while the property was unable to be checked as a composed system due to insufficeint memory. Although to date this technique have been successful only for small examples, by incorporating a more elaborate typing system to partition a system we expect to be able to apply our technique to large scale systems in the near future.

7. CONCLUDING REMARKS

We have investigated several fundamental topics to improve the reliability of network software. In particular, we have been pursuing research topics based on the communicating process model. Although the communicating processes have been studied mostly in the theoretical context, we have been trying to adapt the model for more practical use. In this respect, we are focusing on the reasonable treatment of values and time in communications. These are the fundamental notions in programming, but have often been abstracted away to simplify the theory. Particulary, our first result shows that time is not a simple notion in its behavior and needs to be treated carefully, otherwise, serious errors are likely to occur in network software.

8. REFERENCES

[1] C.A.R Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[2] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[3] J.C.M.Baeten and W.P.Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[4] I. Ulidowski and S. Yuen, "Process languages with discrete relative time based on the ordered sos format and rooted eager bisimulation," *Journal of Logic and Algebraic Programming*, vol. 60–61, pp. 401–460, 2004.

[5] G.D. Plotkin, "A structural approach to operational semantics," Daimi fn-19, Computer Sicence Department, Aarhus University, 1981.

[6] J.F. Groote and F.W. Vaandrager, "Structural operational semantics and bisimulation as a congruence," *Information and Computation*, vol. 100, no. 2, pp. 202–260, 1992.

[7] J.F. Groote, "Transition system specifications with negative premises," *Theoretical Computer Science*, vol. 118, no. 2, pp. 263–299, 1993.

[8] Bard Bloom, "Structural operational semantics for weak bisimulations.," *Theor. Comput. Sci.*, vol. 146, no. 1&2, pp. 25–68, 1995.

[9] Irek Ulidowski and Iain Phillips, "Ordered sos process languages for branching and eager bisimulations.," *Information and Computation*, vol. 178, no. 1, pp. 180–213, 2002.

[10] I. Ulidowski and S. Yuen, "Process languges for rooted eager bisimulation," in *CONCUR 2000*, D. Miller and C. Palamidessi, Eds. 2000, vol. 1877 of *LNCS*, pp. 275–289, Springer.

[11] Xavier Nicollin and Joseph Sifakis, "The algebra of timed processes, atp: Theory and application," *Information and Computation*, vol. 114, no. 1, pp. 131–178, 1994.

[12] Hiroaki Kuwabara, Shoji Yuen, and Kiyoshi Agusa, "Congruence Properties for a Timed Extension of the pi-Calculus," in *Supplemental Volume of the DSN2005*. 2005, pp. 207–214, IEEE Computer Society.

[13] Shoji Yuen, Keishi Kato, Daiju Kato, and Kiyoshi Agusa, "Web automata: A behavioral model of web applications based on the mvc model," *Computer Software*, vol. 22, pp. 44–57, 2005.

[14] A.Mizuno, K.Mano, Y.Kawabe, H.Kuwabara, K.Agusa, and S.Yuen, "Name-passing style gui programming in the pi-calculus-based language nepi," *Electric Notes in Theoretical Computer Science*, vol. 139, no. 1, pp. 145–168, 2005.

[15] D. Sangiorgi and D. Walker, *The Pi-Calculus: A Theory of Mobile Processes*, Cambridge University Press, 2001.