Name-passing style GUI programming in the π-calculus-based language Nepi

Atsushi Mizuno
Graduate School of Information Science, Nagoya University,
1 Furo-cho, Chikusa-ku, Nagoya-shi, Aichi, 464-8603, Japan

Ken Mano
Yoshinobu Kawabe
NTT Communication Science Labs., NTT Corp.,
3-1, Morinosato Wakamiya, Atsugi-shi, Kanagawa 243-0198 Japan

Hiroaki Kuwabara
Kiyoshi Agusa
Graduate School of Information Science, Nagoya University,
1 Furo-cho, Chikusa-ku, Nagoya-shi, Aichi, 464-8603, Japan

Shoji Yuen
Graduate School of Information Science, Nagoya University,
1 Furo-cho, Chikusa-ku, Nagoya-shi, Aichi, 464-8603, Japan
JST PRESTO.
4-1-8, Kawaguchi-honmachii, Kawaguchi, 332-0012, Japan
Abstract

This paper describes name-passing style Graphic User Interface (GUI) programming in the programming language Nepi whose operational semantics is based on the rendezvous-style name-passing communication of the $\pi$-calculus. Nepi is able to have 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 an event to specified name-passing. In the extended Nepi, a GUI program is described as the composition of graphic components, event handling processes, and function processes that implement the real function. We present an implementation of a GUI extension for Nepi programming language on Allegro Common Lisp to illustrate the features of name-passing style GUI programming in Nepi with examples. Finally, we discuss a formal treatment and verification technique based on the extended reduction semantics of Nepi.

Keywords: Network programming languages, graphic user interface, $\pi$-calculus, timed model

1 Introduction

When building a graphic user interface (GUI) of an interactive program, the programmer describes both the graphic appearance and interactive behavior of GUI components such as buttons and textboxes. Since such coding follows specific patterns, many GUI frameworks have been proposed such as Qt for KDE [4], GTK for GNOME [2], and SWING for Java [9]. In many GUI frameworks, a graphic interface is built up by combining GUI components where all reactive behavior is embedded in these GUI components as 'call-back functions'.

Call-back functions are invoked by associated interaction events. The execution of call-back functions differs from the overall execution flow of the program in that call-back functions start and terminate via interaction events from the environment. Embedding codes to handle events for controlling call-back functions often results in meta-level encoding such as passing functions. In analyzing and understanding GUI programs, such non-uniformity of the control level often causes difficulties since the states of the program are difficult to identify.

This paper proposes GUI programming based on event-based characterization via name-passing in the $\pi$-calculus instead of the conventional functional, i.e. input/output, characterization. We aim to provide a fundamental formal framework for analyzing interactive behavior and the execution flow of

1 Email: a-mizuno@agusa.i.is.nagoya-u.ac.jp
2 Email: mano@theory.brl.ntt.co.jp
3 Email: kawabe@theory.brl.ntt.co.jp
4 Email: kuwabara@agusa.i.is.nagoya-u.ac.jp
5 Email: agusa@is.nagoya-u.ac.jp
6 Email: yuen@is.nagoya-u.ac.jp
GUI programs. To present the real effect of our modeling, we extend Nepi programming language [7] capable to describe the interactive behavior of GUI programs. A GUI program is designed as a combination of graphic components and action processes. Graphic components and the associated event handling processes capture events from the environment and then pass the corresponding names. Action processes invoke the corresponding actions associated with the GUI. This modeling reflects the nature of event based behavior in the GUI where the control flow is passed by events for the associated actions.

The name-passing style GUI modeling is briefly described as follows. To create a GUI instance, a name is bound to a GUI component. The basic idea to treat GUI component in name-passing style is shown in figure 1. The extended syntax \( (? \text{g u} (x) P) \) create a GUI component as an instance with class \( u \). At the same time, it binds a fresh name to \( x \) that may appear freely in \( P \). Binding the name to an instance of the GUI component, the name is enabled to interface all activities for GUI functionalities such as open/close operation and various properties such as window size parameters. For this purpose, we introduce a binder to the language in addition to \( \nu \)-operator. The name implicitly communicates only with the GUI environment by receiving a special form of an object name. Further details will be explained in section 3.

Nepi programming language directly describes the name-passing computation of the synchronous \( \pi \)-calculus including timed behavior. The combination of the CCS-like choice operator and wait prefix enables the timeout behavior. We shall add a pair of new syntactic elements to assign a specific name to a graphic component. Via the name assigned to a graphic component, events are passed to invoke methods in function processes where the methods wrap call-back functions. Since Nepi language incorporates Lisp S-expressions for name expressions, we extend the language to bind a graphic component of a Lisp graphic package to a name. We implement the extension on Allegro Common Lisp, “ACL” for short, and the ACL common graphics package. We present the design and implementation of our Nepi language extension by
ACL and illustrate our name-passing based GUI programming by providing a typical GUI example.

To enjoy the inherent advantage of the Nepi language in that the behavior is directly based on the formal name-passing model, we discuss the formal operational semantics of our extended Nepi programs. We present the abstract execution of a Nepi program with the GUI extension. We discuss the way in which the verification techniques are applied to our modeling. Since the GUI environment is outside of Nepi and Nepi interacts with the GUI environment through ACL primitives in a specified manner, we give a description of the GUI environment in Nepi. By attaching the environment to a Nepi GUI program, an execution of GUI program is modeled by labeled transitions.

This paper is organized as follows. In section 2, we briefly review the Nepi programming language and section 3 shows the basic design of our GUI extension. In section 4, we present the extended Nepi language. Section 5 details the mechanism for implementation and Section 6 provides some examples. Section 7 discusses the formal treatment of GUI functions and Section 8 consists of concluding remarks and a brief mention of future work.

2 Nepi Programming Language: Overview

The Nepi Language is designed to describe network programs over TCP-connected networks based on the communicating process model of the π-calculus. The language aims at reliable network programming with the help of the formal semantics of name-passing calculi. In order to describe a 'real' program, the language is extended from the π-calculus in the following two respects: value expressions and stream prefixes. Value expressions ease the handling of predefined data and the control of the conditional branches as in other popular programming languages. Stream prefixes enable the lazy evaluation of stream data such as files. The extension utilizes the LISP syntax for the efficiency reason.

Below is a brief summary of the syntax of Nepi language. More detailed explanations are found in [5][6].

2.1 Value Expressions

A value expression is an S-expression with signature Σ over constants C and variables V. C includes special symbols \{t, nil\} and all names \(C_N\) that are symbols beginning with 'f'. V are symbols beginning with '. Σ is assumed to include all the Lisp built-in functions and the special built-in function ch. ch maps a name to a real TCP port. \((ch \ s_1 \ s_2 \ n)\) is evaluated as the name identifying the TCP connection on port n with the host specified by \(s_2\) with
In the above definition, expressions not to be closed under the structural congruence. 

When a branch of choice is a conditional or a process call, the branch is made the unit for end. Indeed, if end were the unit of \( \text{par} \), the equation \( + P Q = (+ \text{par} P \text{end}) Q \) would be derived, which would make the class of process expressions not to be closed under the structural congruence.

\[ (\text{defproc } p \left( x_1, \ldots, x_n \right) P) \]

\[ (p \ u_1 \ \cdots \ u_n) \equiv F[\underbar{x}_i/x_i] | i = 1, \ldots, n \]

\( (G/ \equiv, +, \text{end}) \) is a commutative monoid. \( \text{par} \) is associative and commutative.

\[ (\text{new } x \text{ end}) \equiv \text{end}, (\text{new } x \text{ (new } P)) \equiv (\text{new } y \text{ (new } x \text{ } F')) \]

\[ (\text{new } x \text{ (par } P Q)) \equiv (\text{par } P \text{ (new } x \text{ } Q)) \text{ if } x \not\in \text{FV}(P) \]

\[ (\text{new } x \text{ (+ } P Q)) \equiv (+ \text{ } P \text{ (new } x \text{ } Q)) \text{ if } x \not\in \text{FV}(P) \]

\[ (\text{if } u \ P_1 \ G_1) \equiv G_1 \text{ if } \text{eval}(u) \neq \text{nil} \quad (\text{if } u \ P_2 \ G_2) \equiv G_2 \text{ if } \text{eval}(u) = \text{nil} \]

\[ \quad \text{Fig. 2. The Nepi language} \]

\[ \quad \text{a nickname of } s_1 \text{. The function } \text{ch} \text{ maps a name to the concrete TCP channel with processes on the remote host. We implicitly assume that the types for data such as 'integer' or 'real' are introduced for value expressions according to LISP built-in functions.} \]

\[ \quad 2.2 \text{ Core Expressions} \]

Figure 2 shows a syntax for core process expressions. For any \( x \in \mathcal{V} \cup \mathcal{C}_N \) appearing in a process expression \( P \), if the occurrence of \( x \) is in the underlined part of a subterm \( (\text{new } x \ Q) \), \( (\text{? } u \ (\cdots \ x \ \cdots) \ Q) \) or \( (\text{?e } u \ (\cdots \ x \ \cdots) \ Q) \) of \( P \), then this occurrence of \( x \) is called bound; otherwise, this occurrence is free.

Although \( + \) is formally a binary operator \( (+ \ P \ Q) \). we write \( (+ \ G_1 \ldots \ G_n) \) with any number of arguments due to the AC nature of operator\( + \). We show the structural congruence \( \equiv \) in figure 3. and the reduction relation \( \rightarrow \) for the core expression in figure 4.

When a branch of choice is a conditional or a process call, the branch is
\[ P \rightarrow P' \]
\[ \langle \text{par} \, (+ \ldots ) \, \text{par} \, P \ldots \rangle \rightarrow P' \]
\[ P_1 \equiv P'_1 \quad P'_1 \rightarrow P'_2 \quad P'_2 \equiv P_2 \]
\[ P_1 \rightarrow P_2 \]

\[ \langle \text{par} \, P_1 \, P_2 \rangle \rightarrow \langle \text{par} \, P'_1 \, P'_2 \rangle \]
\[ (\text{new } x \, P) \rightarrow (\text{new } x \, P') \]
\[ \langle \text{close } (u) \, P \rangle \rightarrow P \]

- In the above, \( c_1 \) and \( c_2 \) are the names for the controls of the stream specified by \( u \).

Fig. 4. SOS for Nepi language

Reduced according to the structural congruence rules. For instance, if there is a process definition \( \text{defproc } a \, (x_1 \ldots x_n) \, P(x_1, \ldots, x_n) \), process expression \( (+ \, (a \, v_1 \ldots \, v_n) \, Q) \) is reduced to \( (+ \, P(v_1, \ldots, v_n) \, Q) \). If a process call is recursively defined by \( \text{defproc} \) with an unguarded body like \( \text{defproc } p \, p \), the behavior of the process may be undefined.\(^8\)

2.3 Extended Expressions

In a Nepi program, a stream channel can be treated in the same way as a usual name. A input prefix with a stream channel binds the head object of the stream, and a output prefix with a stream channel appends an object to the stream. To open a stream for \( w \), we write \( \langle \text{open } (x_1 \, x_2) \, P \rangle \). This binds a newly created name for a stream \( w \) to \( x_1 \) and the size of \( w \) to \( x_2 \). For example, to open a file \( /etc/hosts \), we write \( \langle \text{open } ("/etc/hosts") \rangle \, (x_1 \, x_2) \, P \) \( P \rangle \). Then, a newly created name for the (file) stream is bound to \( x_1 \). \( x_2 \) is bound to a new name to obtains a control information such as the file size. To close a stream, we write \( \langle \text{close } (u) \, P \rangle \) where a stream name \( u \) is closed and release the resource for the stream. Both prefixes are reduced by \( \rightarrow \) autonomously. The further details for the stream prefixes are found in [5][6].

During the course of designing Nepi, the stream prefixes are introduced to treat files. It inspires us to give a general syntax to create a name with extra functionalities. \( ?e \) gives a name with the file stream functionality. Subsequently we propose a similar binder to give names with the GUI functionality.

2.4 Timed behavior

To describe the timed behavior, Nepi has a timed prefix of form: \( \langle \text{time } n \, P \rangle \) that enables to make \( P \) wait for \( n \) seconds before starting. Here \( n \) can be

\(^8\) The interpreter may try to infinitely expand the process resulting in infinite loop. But it is just an abnormal behavior and differs from the infinite sequence of reductions.
any value expression. At the moment, in order to behave as intended, \( n \) has to be a value expression evaluated as a non-negative integer, referred to as 'Number' in Nepi. If it is evaluated as other than a non-negative integer, the whole expression is evaluated as an error.

Figure 5 shows the timed transition semantics. The time tick is a reduction labeled by '1'. \( P \overset{1}{\rightarrow} P' \) means that \( P \) evolves to \( P' \) after one unit of time passage. As the timed prefix is categorized as a guarded expression, it enables time-out behavior to be combined with the choice operator. The time tick \( P \overset{1}{\rightarrow} P' \) decreases the timer counter. When the time expires to zero, it makes a silent transition. Please note that this silent transition resolves the choice where the time-out choice is achieved. For instance, \((+ (? \ Chan \ (x) \ P) \ (\text{time} 5 \ Q))\) behaves as \( P \) (with the appropriate replacement of \( x \)) when it receives a name via \( \text{chan} \). If there is no communication within 5 seconds on \( \text{chan} \), it reduces to \((+ (? \ Chan \ (x) \ P) \ (\text{time} 0 \ Q))\). At this point, if no communication available on \( \text{chan} \), by making the choice on the second summant it behaves like \( Q \) discarding the first choice.

The basic timed properties (up to the structural congruence) can be checked as follows for the time determinacy, the maximal progress and the weak-timelock freeness[10]. The time determinacy holds since there is no 'dynamic' operator with respect to the timed reduction. The maximal progress holds because of the precondition of \( \text{par} \)-rule. The weak-timelock freeness holds since there is no urgent operator other than \((\text{time} 0 \ P)\). There may be a possibility to lock the system by the infinite reduction.

3 GUI Modeling by Name-Passing

We model a GUI component observed as the composition of the following three types of communicating processes.

(i) **Graphic component**: A process for representing the appearance of a GUI component, such as buttons. It has appropriate properties such as size or color. A graphic component knows the names of relevant event-
handling processes. The environment manages the creation and termination of a graphic component by passing names.

(ii) **Action process:** A process that does the real job. It corresponds to the call-back function in usual GUI programming. Upon receiving the appropriate name from the associate event-handling process, it actually reacts to the input and communicates with the processes.

(iii) **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 from an action process.

An overview of the behavioral model is shown in figure 6. 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. 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 action process. In this respect, the typical name passing facility is used. After this initial procedure is complete, every time an event is accepted the event-handling process sends this information to the action process followed by some reaction to the graphic component.

After creating a graphic component it creates event-handling processes 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 to the corresponding names. When closing a graphic component, a privileged name has to be sent to the component to kill all sub-components.
4 Nepi Language Extension for GUI

In this section, we present the extended syntax and its behavior for our GUI modeling in the Nepi language. We introduce a pair of syntax elements \(?g\) and \(!g\) that create and terminate a graphic component, respectively. When a \(?g\)-prefix is reduced, a newly created channel is bound to a graphic component. When a \(!g\)-prefix with close is reduced, the graphic component is terminated.

4.1 Creation and Termination of Graphic Components

We add the following prefixes to figure 2 to manipulate the graphic components:

\[
P ::= \ (\ ?g\ u\ (gch)\ P) \quad \text{creation} \nonumber \\
| \ (\ !g\ close\ (v)\ P) \quad \text{termination}
\]

where \(u\) is a Lisp object that defines the appearance of the graphic component (usually generated by the ACL Interface Builder), \(v\) is a value expression, and \(gch\) is a variable that is bound to a fresh name created for the graphic component.

Note that \(?g\)-prefixes and \(!g\)-prefixes are categorized as process expressions despite their syntax. This is because graphic manipulation is inherently not an external event for controlling the behavior.

**Creation of graphic component:** \((?g\ u\ (gch)\ P)\) receives a GUI channel \(c\) with an appearance specified by \(u\) from the environment. The free occurrence of \(gch\) in \(P\) is replaced with \(c\). The newly created graphic component \(gc\) holds the GUI channel \(c\) to communicate with the process.

Since \(gch\) has to be “ground” in that all the sub-components are instantiated, if \(u\) has uninitialized sub-components, then for each sub-component \(u',\ (?g\ u'\ (gch)\ (!u\ ((\text{list add-graphic-component}\ u'))\ end))\) is recursively created, where \text{add-graphic-component} message adds a graphic sub-component of \(u'\).

**Termination of graphic component:** \((!g\ close\ (v)\ P)\) evaluates the value expression \(v\) to have a GUI channel \(c\), where \(c = \text{eval}(v)\). It sends \(c\) to the environment via a well-known channel close, and then performs the continuation \(P\). The environment sends a termination signal to the corresponding graphic component \(gc\) via \(c\) if \(gc\) exists. The graphic component \(gc\) terminates after receiving the termination signal. As regards \(?g\), for each sub-component of \(gc\) with GUI channel \(c'\), \((!g\ close\ (c')\ end)\) is recursively created to terminate all sub-components.
4.2 Requests to Graphic Components via GUI Channels

Once a graphic component is created, a user program may send various kinds of requests to the graphic component. An output process expression in the form \((! \text{gch} \ (\text{req} \ \text{ret-ch}) \ P)\) represents the process that sends a request to a graphic component bound to \(\text{gch}\). Here \(\text{req}\) is a list consisting of a request directive followed by its arguments (possibly none), and \(\text{ret-ch}\) is the name used for sending the value obtained by handling the request to the action process. Depending on the form of \(\text{req}\), the graphic component bound to \(\text{gch}\) is manipulated by the directive.

**Getting property value** A property symbol is associated with each property of a graphic component. Typical property symbols are shown in Table 1. For a property symbol \(\text{prop}\), a list \((\text{prop})\) is a request to send the current property value. For instance, \((\text{background-color})\) is a request for the value of the background color property. When a graphic component receives a request \((\text{prop})\), it sends the current value of the property \(\text{prop}\) to \(\text{ret-ch}\).

**Setting property value** A request for setting a property value is a list with the following form: \((:\text{set} \ \text{prop} \ v)\). Here \(\text{prop}\) is a property symbol and \(v\) is a value expression. When a graphic component receives this request, it updates the value of the property \(\text{prop}\) to \(v\), and sends \(v\) via \(\text{ret-ch}\).

**Creating event handling process** An event symbol is associated with each event. Typical event symbols are shown in Table 2. \((:\text{event} \ \text{event} \ \text{event-ch})\) is a request to create an event handling process for \(\text{event}\). If a graphic component receives this request, it then tries to create an event handling process, and sends \(t\) if this succeeds and \(\text{nil}\) otherwise. Each time an event occurs, the created event handling process sends values concerning the event via the name \(\text{event-ch}\).

**Graphic Commands** Graphic commands sends a request for the graphic component itself. Typical commands are shown in Table 3 (the command \(\text{close}\) is intended to be used only by the environment, and not by action processes). The request for the command represented by the symbol \(\text{command}\) is in the form: \((\text{command} \ v_1 \ \cdots \ v_n)\).

4.3 Event Notification

An event handling process is created by sending an \(\text{:event}\) directive to a graphic component for each event to be captured. For instance, a prefix \((! \ u \ ((\text{list} \ :\text{event} \ \text{on-click} \ c) \ \text{ret-ch}) \ P)\) creates an event handling process that is ready to send a signal on \(c\) when the raw event of ‘on-click’ occurs. Table 4 shows the correspondence between the raw events and the values sent
Table 1
Properties of Graphic Component

<table>
<thead>
<tr>
<th>Property symbol</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>title</td>
</tr>
<tr>
<td>height</td>
<td>height</td>
</tr>
<tr>
<td>width</td>
<td>width</td>
</tr>
<tr>
<td>background-color</td>
<td>background color</td>
</tr>
<tr>
<td>foreground-color</td>
<td>foreground color</td>
</tr>
</tbody>
</table>

Table 2
Events of Graphic Component

<table>
<thead>
<tr>
<th>Event symbol</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>on-change</td>
<td>change of value's value</td>
</tr>
<tr>
<td>on-click</td>
<td>click</td>
</tr>
<tr>
<td>on-double-click</td>
<td>double click</td>
</tr>
<tr>
<td>on-mouse-in</td>
<td>entrance of mouse pointer</td>
</tr>
<tr>
<td>on-mouse-out</td>
<td>exit of mouse pointer</td>
</tr>
</tbody>
</table>

Table 3
Commands of Graphic Component

<table>
<thead>
<tr>
<th>Command symbol</th>
<th>Command</th>
<th>Arguments</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>close graphic component</td>
<td>none</td>
<td>t</td>
</tr>
<tr>
<td>set-keyboard-focus</td>
<td>get keyboard focus</td>
<td>none</td>
<td>GUI channel</td>
</tr>
<tr>
<td>add-graphic-component</td>
<td>adopt a graphic component</td>
<td>GUI channel</td>
<td>GUI channel</td>
</tr>
<tr>
<td></td>
<td>as child</td>
<td>of child</td>
<td>of child</td>
</tr>
</tbody>
</table>

Table 4
Values Sent from Event Handling Process

<table>
<thead>
<tr>
<th>Event symbol</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>on-change</td>
<td>value's old and new values, termination flag</td>
</tr>
<tr>
<td>on-click</td>
<td>clicked graphic component’s GUI channel, termination flag</td>
</tr>
<tr>
<td>on-double-click</td>
<td>double-clicked graphic component’s GUI channel, termination flag</td>
</tr>
<tr>
<td>on-mouse-in</td>
<td>mouse buttons’ status, GUI channel of window that mouse pointer was previously in, termination flag</td>
</tr>
<tr>
<td>on-mouse-out</td>
<td>mouse buttons’ status, GUI channel of window that mouse pointer entered next, termination flag</td>
</tr>
</tbody>
</table>

back via the return channel.

5 Implementation

In accordance with earlier implemented versions of the Nepi language the GUI extension is implemented using the Common Graphics package of ACL. We briefly explain GUI programming with the Common Graphics package by using an example, and then we describe several important implementation issues in relation to GUI.
5.1 ACL Common Graphics package

In Common Graphics, GUI components such as a window or a button are modeled as an object called a widget with properties and methods. For instance,

\[
\text{make-instance 'button} \\
\text{ :top 100 :left 100 :value 'OK} \\
\text{ :on-change nil :on-click nil)
\]

creates a button widget. top, left and value are properties of the object, and 100 and OK are values of the properties. Given a widget wid and property prop, the value of prop held by wid is obtained by evaluating (prop wid). The property is updated to val by evaluating (setf (prop wid) val).

For each widget, detectable events are defined by Common Graphics, and event names are also treated as properties of a widget. Given a widget wid with a detectable event ev, by setting the method name as ev, the method is invoked when ev occurs to wid.

5.2 Implementation of GUI function

As regards the whole Nepi system, a Nepi processor is running on each host and the hosts are connected to each other over the TCP network. When a Nepi processor is started on a host, a passive socket is generated. Each Nepi processor has an identifier called address that is a pair consisting of an ip-address and a port number over TCP. The address is used for communication with other Nepi systems and for fresh name creation. Along with the configuration of the Nepi system, each name has the attributes of the host information, the channel type: REG (regular name), STR (stream data channel), or GUI (GUI channel).

Process expressions are held in a memory area called process ring in a Nepi system on each host. The system obtains a process expression from the process ring and interprets the expression to be reduced.

5.2.1 Graphic component manipulation

Creation:

If a process expression (?g u (gch) P) is in the process ring, a widget wid and the corresponding GUI component are created.

A new channel structure object for GUI is obtained by Nepi’s fresh name creation function on the local host. A triple (c wid nil) is then added to the graphic component table, and finally each free occurrence of gch is P is replaced by c where nil is the initial empty list for handling events.
Termination:

Let the process expression be $(!g \text{ close } (u) \ P)$. There are two cases, depending on the channel $c$ obtained by the evaluation of $u$.

If $c$ is a channel on the local host, the system searches the graphic component table for the concrete window object and the event-handling list. Before sending the close signal to $c$, the process seeks the graphic subcomponents to emit the termination signal recursively by issuing $!g$-prefixes continued by $\text{end}$ to the process ring.

If $c$ is a channel on the remote host, the local Nepi processor sends out a process expression $(!g \text{ close } (c) \ \text{end})$ to the process ring of the remote system.

5.2.2 Processing requests

Requests to GUI components are processed by reducing the prefixes whose subjects are GUI channels created by $?g$-prefixes. Note that a prefix appears as the first element of a guarded process expression. Although the abstract semantics of the choice operator $+$ follows that of the conventional choice, the Nepi system tries the guards from left to right, for practical reasons related to implementation. (For the detailed implementation mechanism, see [3].)

Suppose a GUI output $G_i \equiv (! \ ch \ (\text{req ret-ch}) \ P_i)$ is chosen. When $ch$ is evaluated as a local name $c$, the Nepi processor sends the request to the graphic object for $c$ by looking up in the graphic component table. If $ch$ is evaluated as a name on a remote host, the local Nepi processor dispatches $(! \ ch \ (\text{req ret-ch}) \ \text{end})$.

Getting and Setting Property values Property values are obtained or set simply by translating the channel to the corresponding window object by looking up in the graphic component table. For instance, $(! \ ch \ ((\text{list title}) \ ret) \ P)$ sends the graphic object $win$ corresponding to $ch$ if the triple $(ch \ win \ alist)$ exists in the table. Property values are set by $\text{setf}$ Lisp macro. If any error should occur, a special value $\text{err}$ returns on $ret$.

Creating event-handling process Suppose $req$ is a request to create an event-handling process, that is, $(: \text{event ev ev-ch})$ is obtained by evaluating $req$ for an event symbol $ev$ and a name $ev$-$ch$. Then the system evaluates $(\text{setf } (ev \ wid) \ func_{ev})$. With the Common Graphics event facility, the function $func_{ev}$ is invoked when an event $ev$ occurs for $wid$. Such a function $func_{ev}$ is defined in the Nepi system for each $ev$.

Graphic commands If $req$ is a request for the execution of a command, the Nepi processor simply executes the command with the corresponding windows as the first argument; $(\text{command wid } v_1 \cdots v_n)$.
It should be noted that handling the graphic sub-components mentioned in 4.1 is implemented as a command request:

\[
\text{(! } gch_1 ((\text{list add-graphic-component gch}_2) \text{ ret-ch}) \text{ end)}
\]

The system evaluates \(\text{list add-graphic-component gch}_2\) and obtains a list \(\text{add-graphic-component gch}_2\). It then finds an entry \((gch_1 wid_1 alist_1)\) in the correspondence table, and evaluates the expression

\[
(\text{add-graphic-component wid}_1 gch_2)
\]

using a function \text{add-graphic-component} supplied by Nepi. As a result, the widget corresponding to the GUI channel \(gch_2\) is added to the child widget list of \(wid_1\).

5.3 Event Notification

As described above, when an event \(ev\) that is registered at the creation of the event-handling process occurs for a widget \(wid\), an expression \((func_{ev} wid \ v_1 \cdots v_n)\) is evaluated. Here, \(v_1, \ldots, v_n\) are values associated with the event. The function \(func_{ev}\) tries to find a triple of the form \((ch \ wid \ alist)\). Suppose \((ev \ . \ ev-ch)\) is the first element in \(alist\) whose car part is \(ev\). Then

\[
\text{(! ev-ch (v}_1 \cdots v_n \text{ nil) end)}
\]

is added to the process ring.

Other than closing with \(! g\), a widget \(wid\) can be closed by clicking the “x” button on the title bar if \(wid\) is a window. In such a case, \((\text{close wid})\) is evaluated automatically by Common Graphics, which results in the invocation of before hook of \text{close} described above.

6 Example

We present a file copier that copies a file to another with the progress bar. First, the program displays the main window (figure 7) on the screen. This window has a start button and two input forms. When the start button is pressed, the program creates a progress-bar window (figure 8), and starts the procedure to copy 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.
6.1 Building GUI with IDE of ACL

We use IDE, the GUI builder of ACL, to design the graphic components. IDE has a graphical editor for designing a window, and it generates a corresponding S-expression. In the file copier example, IDE generates a Lisp program ‘copy.bil’ for the main window, in which the top-level function is ‘make-copy’. A Lisp program ‘progress.bil’ for the progress-bar window is also generated. When the value expression \(\text{make-copy}\) is evaluated, the Nepi system displays the main window on the screen and returns a widget ‘\(\text{widget}_0\)’. Then, the value \(\text{widget}_1 \cdots \text{widget}_5\) is set to the property ‘dialog-items’ of \(\text{widget}_0\), where each GUI component in figure 7 has a number \(i \in \{1, \ldots, 5\}\) that corresponds to \(\text{widget}_i\). Programmers can use the expression \(\text{make-copy}\) in the underlined part of the form \((?g \text{ (make-copy) (gch}_0 \text{ gch}_1 \cdots \text{gch}_5) P)\). Note that we can write six GUI channels \(\text{gch}_0, \ldots, \text{gch}_5\) in this form, which means that this form is an extension of the \(?g\)-expression. This extended form is implemented by creating fresh names \(\text{gch}_1, \ldots, \text{gch}_5\) and assigning them to widgets \(\text{widget}_1, \ldots, \text{widget}_5\), respectively.

6.2 Action Processes for File Copier

Figure 9 is the program code for the file copier. This program has four process definitions ‘nepicopy’, ‘start-handler’, ‘copy-handler’ and ‘copy-handler1’:

- The top-level process nepicopy displays the main window. This process sends a message \((:\text{event on-change $\text{start-ch}$})\) to the start button’s GUI channel, where $\text{start-ch}$ is a new name employed for event notification.
- The process start-handler is an action process for the main window. When the start button is pressed on the main window, this process does the following. First, start-handler reads filenames from two input forms on the main window, and uses them to replace the variables ‘\(.\text{from-file}\)’...
and `.to-file`. Then, this process tries to open the files. When there is a failure, the process displays a pop-up window providing the notification of the failure. After opening these files, the process creates a progress-bar window, and generates an event-handling process for the progress-bar window’s cancel button. This process times out in 60 seconds. If no event on `.start-ch` occurs, it closes the window and its sub-components die.

- The process `copy-handler` is an action process for the progress-bar window; `copy-handler1` is a sub-routine of `copy-handler`. This process is called by `start-handler`, and tries to perform one of the two following actions:
  - The first action is to receive a message from the cancel button’s event-handling process through the name `.cancel-ch`. If the message contains the value `t` for the variable `.fin`, then the process terminates. Otherwise, the process closes the progress-bar window;
  - The second action is to read a line from the stream channel `.input`. If the line is EOF, then the progress-bar window is closed. Otherwise, the process attaches the line to the output file, and then, some property values are changed with regard to the progress-bar window.

7 Discussion

This section discusses the formal operational semantics of GUI behavior in Nepi programs. We define a basic transition system taking into consideration the graphic objects created and destroyed during the computation.

7.1 Operational Semantics of GUI Behavior

We assume a reserved set of channels for widgets, `Gch`. A closed process expression having some names of `Gch` is called a GUI system.

Definition 7.1 A Graphic Context is recursively given in the following way:

- `[]` is a graphic context;
- `<G, ⟨v : u⟩>` is a graphic context when `G` is a graphic context; and
- `νv.(G, ⟨v : u⟩)` is a graphic context.

where `v` is a name and `u ∈ Gch`.

We write `G[ P ]` when `[]` is replaced by a process expression `P` in `G`. A set of free names in `G`, `fn(G)` is defined as follows: `fn([]) = ∅`; `fn(<G, ⟨v : u⟩>) = fn(G) ∪ {v}`; and `fn(νv.(G, ⟨v : u⟩)) = fn(G) − {v}`. The Nepi operational semantics[6] is extended for graphic labels as shown in figure 10. The transitions relevant to the GUI manipulation are labeled by `Gch` to reflect the effect
of the graphic appearance. Intuitively, \( G[P] \) represents an action process \( P \) in a graphic appearance of \( G \). A transition by \( ?g \) creates more graphic objects in \( G \) and \( !g \) terminates graphic objects in \( G \).

The GUI behavior is the transition relation \( \alpha \rightarrow g \) as defined in figure 11. \( G[P] \xrightarrow{\alpha} G'[P'] \) means that process \( P \) with the graphic appearance of \( G \) evolves to \( P' \) with the updated graphic appearance of \( G' \) by the effect of \( \alpha \).

An execution of a GUI system begins with the empty graphic context followed by the sequence of \( \tau \rightarrow g \) changing the graphic appearance. Although \( G \) does not directly hold the screen image, we record the type of graphic object. We consider an execution can hold the sufficient information to distinguish the graphic appearance.

**Definition 7.2** Given a GUI system \( P \), an execution of \( P \) is a (possibly infinite) sequence of \( \tau \rightarrow g \) and \( 1 \rightarrow g \) beginning with \( P \).

\[
P = P_0 \xrightarrow{\alpha} g \ G_1[P_1] \xrightarrow{\alpha} g \ G_2[P_2] \xrightarrow{\alpha} g \cdots
\]

where \( \alpha \) is either \( \tau \) or \( 1 \).

In the execution, only \( \tau \) or \( 1 \) may appear in the transitions. \( P \) is intended to be given in the form \( \text{par U Env} \) where \( U \) is a user program and \( Env \) is the environment that is common to all user programs. The user program \( U \) operates under the environment \( Env \). In the above formalization, \( Env \) has to include all the patterns of interaction with the outside in order to represent the complete set of executions. Such an environment is unable to express in the form of pure process expression in general. With the help of the Lisp built-in function, we can obtain a reasonable approximation for the complete environment as shown in the next section.

7.2 **The Environment Description**

From the more concrete view of modeling the behavior of the GUI program, we show the environment in the Nepi language using the Lisp built-in functions.
Here \( \text{exp-1}, \text{exp-2}, \ldots \) is a sequence of all expressions representing graphic components. Although this summation can be infinite in general, given a specification of user programs, the set of classes appearing in the programs is fixed.

The process generating an instance of \( u \) is defined as a process \( \text{GEnv}_u \) parameterized by \( u \). Process \( \text{GEnv}_u \) creates a fresh GUI channel \( \$gch \) and sends it to \( \text{close-gc} \). Upon emitting \( \$gch \) via \( u \), the process starts the corresponding graphic component \( \text{(NewGraphicComponent}_u \$gch) \). The process handling close requests is given as \( \text{close-gc} \). The process accumulates the names of graphic components received via \( .open \) in the list \( .open-lst \). Upon receiving a close request, the process forwards it to the corresponding graphic component and removes the name from \( .open-lst \).

The environment description above is reminiscent of the object description by Walker in the \( \pi \)-calculus [11] where an instance creation is modeled as passing private names to invoke the instance. Since Nepi does not have the replication operator, we write it with the recursion by \text{defproc}.

7.3 Towards Verification of Behavioral Properties

As stated in the preceding sections, in order to verify a user GUI program in Nepi, we have to place the program in the environment to check its execution. The environment description in the previous section provides a reasonable
counterpart for a given user program with respect to Nepi implementation. Since the interaction pattern with the outside is not finitely specified, we cannot verify the behavior of the GUI program. However, by limiting the interaction pattern, it is possible to verify a property with respect to the interaction pattern.

For example, the graphic component ‘prog-window’ is defined as:

```lisp
(defproc prog-window (.gch .progress .cancel .val .text)
 (? .gch (req ret-ch)
  (if (eq (first req) close)
   (! .progress ((list close))
   (! .cancel ((list close))
   (! .val ((list close))
   (! .text ((list close)) end))))
  (prog-window .gch .progress .cancel .val .text))))
```

to specify a progress bar in Section 6. This graphic component receives a command ‘close’ through a GUI channel ‘.gch’ and sends a command ‘close’ to its child widgets through channels ‘.progress’, ‘.cancel’, ‘.val’ and ‘.text’. And then, the graphic component terminates. Similarly, other processes that constitute the GUI system can be described in Nepi.

By checking the executions, we can discuss the property of the program. For example, the following property should hold for the program in Section 6:

**Property:** After the start button is pressed, if a cancel button on a progress-bar window is pressed, then the window will finally be closed.

This property can be restated by saying that all the executions terminate with the empty graphic context. Depending on the race between copy function and cancel, the window may be closed by the end of copy even if the cancel button is pressed. To prove this property, we only have to consider the limited environment for the pattern of events in the regular expression: “(From|To)*.Start.(Progress|Text|Val)*.Cancel” where ‘From’ and ‘To’ are the events to input the files names, ‘Start’ is to press the start button, ‘Progress’, ‘Test’, ‘Val’ are the events to show the progress bar, and ‘Cancel’ is to press the cancel button. Such an environment can be expressed in a finite way. We have not developed any concrete technique yet. We expect such limited environments can be specified by expressing the environment as the combination of the experiments to be performed and the common environment.

In the sense that the above discussion of ‘verification’ is relative to the idealized standard environment, it can be regarded as the ‘validation’ under such an idealized situation. In order to directly verify behavioral properties, it is necessary to include the whole environment into Nepi to make the behavior under the full control within the system. Theoretically this should be possible, but in the practical sense, since Nepi is designed to give a high-level description combined with ACL, we consider checking properties at this level may well be
a verification of Nepi programs.

7.4 Related Work

Nomadic Pict [13] is a programming system based on the $\pi$-calculus. It can handle the graphics library of Objective Caml [12]. In the modeling for Nomadic Pict, a name represents an interface for a basic operation such as drawing a line or a circle, opening a screen, or detecting a mouse click. For example, in order to draw a line, the positions of the two ends are sent through a name ‘lineto’; this name corresponds to a line drawing operation. Nomadic Pict’s GUI library does not have widgets, so the programmer should write a code for the widgets him/herself. In contrast, by assigning a name to a graphic component, we can employ widgets of Common Graphics in Nepi.

Concurrent ML [8] is a concurrent extension of Standard ML, which provides synchronous communication, name creation, and non-deterministic choice in the same way as Nepi. It has a GUI library called eXene [1] with various widgets described in Concurrent ML. The modeling of eXene, however, strongly depends on the rendering model and event routing implementation of the X window system [14]. On the other hand, Nepi’s modeling is based on an object modeling by Walker[11], which is more abstract and architecture-independent comparing to that of Concurrent ML. Therefore, we expect this approach is to be successful for other object-oriented GUI systems.

8 Concluding Remarks

We proposed name-passing style GUI programming in Nepi programming by extending a pair of prefixes $?g$ and $!g$. We extended the Nepi language processor on Allegro Common Lisp. In the name-passing style, a GUI program is composed of graphic components and action processes. Graphic components accept external events such as clicking buttons to issue names to pass to the action processes. The behavior of a GUI program is modeled as a transition system based on the operational semantics of Nepi language [5]. In our model, there is no need to incorporate the higher-order setup used in the conventional functional treatment. By observing the communication between those two types of components, we can abstract detailed information of graphic components. As the result, it is not necessary to modify the action processes when the change only occurs in the graphic components. This contributes portability to GUI programming.

The introduction of $?g$ and $!g$ expressions is our extension of Nepi where $?g$ assigns a unique name to a graphic component and $!g$ closes a graphic component so that both associated action processes and the graphic appear-
ance disappear. Through the assigned name, the environment can obtain the names for handling events and reading and setting attributes. A similar extension (?e expression) has been made to allow Nepi to deal with streamed data.

We presented the basic operational semantics of ?g and !g expressions. We presented an abstract execution of a GUI program accounting for the graphic appearance in terms with the creation and termination of GUI components. Although the basic behavior of Nepi is defined by the reduction, the GUI behavior inherently has a side effect outside the system. Thus, we extended the behavior by labeled transitions. Nepi also has the capability to deal with discrete timing. Although the timed semantics is not thoroughly investigated yet, the timed extensions for existing process calculi [10] are to be applied. Our timed model does not consider the uncertainty of the time passage needed to communicate with the remote host. At the current stage, our timed behavioral characterization is limited to the time-out behavior on local processes.

Acknowledgements

The authors thank for the valuable comments and suggestion by the anonymous referees. The authors also thank for Yoshifumi Manabe in NTT CyberSpace Laboratories and Eiichi Horita in NTT Information Sharing Platform Laboratories for their helpful discussions. This research is partially supported by the Japanese Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (B)(2)#14380141, Scientific Research(C)(2)#13680408, and Scientific Research(C)(2)#16500027. It is also partially supported by the 21st century COE program of Intelligent Multimedia Integration at Nagoya University.

References


Fig. 9. Program code for a file-copier
\[(\forall u \ (gch \ P) \xrightarrow{u} P[c/gch]\]
\[(\forall \cdot \cdot \cdot \ \forall u \ (c) \ \cdot \cdot \cdot ) \xrightarrow{u} P\]
\[P \xrightarrow{\text{new} \ v \ P} P'\]
\[(\forall \cdot \cdot \cdot \ (\forall c \ (z) \ \cdot \cdot \cdot ) \xrightarrow{c} P[v/z]\]
\[P \xrightarrow{\text{close} \ v \ P} P'\]
\[P \xrightarrow{\text{new} \ v \ P} P'\]
\[P \xrightarrow{\text{close} \ v \ P} P'\]
\[P \xrightarrow{\text{new} \ v \ P} P'\]
\[P \xrightarrow{\text{close} \ v \ P} P'\]

- In the above, \( u \in \text{Gch}, u \neq \text{close}, \) and \( \eta = uc, \overline{uc}, \overline{uc(c)}, \text{closev}, \overline{\text{closev}}, \text{or} \overline{\text{closev}}. \)

Fig. 10. SOS Extension of GUI manipulations

\[\frac{F \rightarrow F'}{G[F] \xrightarrow{\text{new} \ v \ P} G[F']}\]
\[\frac{P \xrightarrow{\text{close} \ v \ P}}{G[F] \xrightarrow{\text{close} \ v \ P}}\]
\[\frac{P \xrightarrow{\text{close} \ v \ P}}{G[F] \xrightarrow{\text{close} \ v \ P}}\]

- In the above definition, \( n(uc) = n(uc(c)) = n(uc(c)) = \epsilon \) where \( u \neq \text{close} \).

Fig. 11. SOS for GUI behavior