Ricardo Silva and Luís Lopes

Technical Report Series: DCC-2006-02

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

---

Universidade do Porto

Rua do Campo Alegre, 823
4150 Porto, Portugal

Tel: +351+2+6078830 – Fax: +351+2+6003654
http://www.dcc.fc.up.pt/Pubs/treports.html

A Debugger for a Programming Language Based on a Process-Calculus

Ricardo Silva and Luís Lopes
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
e-mail:aquavlis@yahoo.com,lblopes@dcc.fc.up.pt

Abstract

This paper describes the implementation of a debugger module for a programming language based on a process calculus. The language was used as a teaching tool for programming with message passing concurrent systems and in this context the development of a debugger for end users was of prime importance to increase usability of the system. We start by describing the syntax and semantics of the programming language and some programming examples. Then, we describe the implementation of the compiler and of the run-time system. Finally, we show how we can implement a debugger in a highly flexible way as a new module for the language.

keywords: Process-Calculus, Programming Language, Compiler, Run-Time System, Debugger.

1 Introduction

An adequate formal model to reason about interacting processes has long been a fundamental goal for programming language designers. The first steps in this direction were given by Milner [17] with the development of CCS (Calculus of Communicating Systems). CCS describes computations in which concurrent processes may interact through simple synchronization, without otherwise exchanging information. Allowing processes to exchange resources (e.g., links, memory references, sockets), besides synchronizing, considerably increases the expressive power of a formal system. Such a system should be able to model the mobility patterns of the resources and thus provide more insight into the system.

The first such model, built on Milner's work, was the π-calculus [19]. Later developments of this initial proposal allowed for further simplification and provided an asynchronous form of the calculus [12, 3]. Since then, several calculi have been proposed, mostly based on the asynchronous π-calculus, to address specific concerns such as security (Spi [1]), to simplify the programming of certain patterns (TyCO [25], Join [9], Klaim [8]) or proposing a different model of interaction (Fusion [21]). Some of these calculi have been used as the basis for the implementation of new programming languages using the mobility paradigm. Examples of such languages are Pict [6], Join and Jocaml [10, 5], XKlaim [15] and TyCO [25].

These languages are mostly prototypes with scarce software development tools available. In particular, specific IDEs (Integrated Development Environments) are mostly unavailable. The tools available for programming and debugging are those of the base language in which the system has been implemented (Java, OCaml, C/C++). This is inadequate for users wishing to learn to program in one of these languages. They should not be given the details of the underlying implementation when investigating a bug but rather the higher level abstract view given by the language computational model.

The TyCO programming language, based on the namesake calculus, was used for introducing the mobility paradigm of process calculi to students. This prompted the need for a more friendly programming environment, namely an IDE for TyCO. The first step in the development of the environment was a debugger for TyCO that provided users with the proper abstraction level for investigating bugs in their programs. By this we mean that the information reported and operations allowed by the debugger should be mapped on to the abstractions of the language itself: channels, objects, method invocations and procedure calls. This provides an adequate tool for investigating semantic errors in TyCO programs and provides a very useful training tool.

The debugger was developed assuming an architecture similar to `gdb` [11]. The idea behind this approach was to allow a seamless embedding of the debugger into a graphical interpreter platform such as `xxgdb` [28] or `ddd` [7]. This would provide a fast prototype for a high level debugging interface for TyCO.

The remainder of this paper is organized as follows. Section 2 introduces the TyCO programming language and its operational semantics. This is followed by section 3 that describes the compilation scheme and the run-time system for TyCO. The next two sections describe the implementation of the debugger, in particular, its integration with the compiler and run-time system. Section 5 describes some work on programming languages based on process calculi from the point of view of the availability of debugging tools. The paper ends with some conclusions and references for future work.

2 The TyCO Calculus and Language

This chapter introduces the TyCO programming language, its syntax and semantics. The operational semantics is given in the form of a reduction relation (term re-writing rules). The static semantics of the language is provided by a type-inference system (not discussed here) which supports a few builtin types, recursive types, and predicative polymorphism *à la* Damas-Milner over process abstractions. The type-system infers the types for the free variables of processes in TyCO programs.

2.1 The Syntax

We introduce some syntactic categories required in the definition of the TyCO language. A sequence $\alpha_1 \cdots \alpha_n$ $(n \geq 0)$ of elements of the syntactic category α is denoted $\tilde{\alpha}$. *Channels*, representing *links*, *references*, *sockets*, are the mobile units of the language, the items exchanged by processes. They are ranged over by a, b. Channels are introduced in processes implicitly with the creation of each new local variable. Boolean, integer or string *constants* are ranged over by c. *Variables* stand for channels or constants in a given process context. They are ranged over by x, y. *Values* are either constants or variables and are ranged over by u, v. *Expressions*, ranged over by e, are builtin logical, algebraic, relational and string operations.

Other kinds of identifiers occur in processes. *Procedures*, ranged over by X, identify processes parameterized in a set of variables. They are introduced syntactically in procedure definitions. *Labels*, ranged over by l, are used to select methods within objects.

Finally, *Processes* are ranged over by P. Their structure is given by the following grammar:

P	::=	**inaction**	Terminated Process
	|	**new** $\tilde{x}$ P	New Local Variable
	|	**def** $X_1(\tilde{x}_1) = P_1 \ \ldots \ X_n(\tilde{x}_n) = P_n$ **in** P	Procedure Definition
	|	$x?\{l_1(\tilde{x}_1) = P_1 \ \ldots \ l_n(\tilde{x}_n) = P_n\}$	Object
	|	$x\,!\,l[\tilde{e}]$	Asynchronous Message
	|	$X[\tilde{e}]$	Procedure Call
	|	$P \mid P$	Concurrent Composition
	|	(P)	Grouping
	|	**if** e **then** P **else** P	Conditional Execution
	|	**io** $!$ **put**$[e]$	Output
	|	**io** $!$ **get**$[x]$	Input
M	::=	$\{l_1(\tilde{x}_1) = P_1 \ \ldots \ l_n(\tilde{x}_n) = P_n\}$	Methods
D	::=	$X_1(\tilde{x}_1) = P_1 \ \ldots \ X_n(\tilde{x}_n) = P_n$	Procedures
e	::=	$e_1 \ op \ e_2 \quad \mid \quad op \ e \quad \mid \quad (e) \quad \mid \quad v$	Expressions

Some syntactic conventions are assumed for processes, namely: a) the parameters $\tilde{x}$ in methods and procedures are pairwise distinct; b) the labels l_i are pairwise distinct in a method collection $\{l_1(\tilde{x}_1) = P_1 \ \ldots \ l_n(\tilde{x}_n) = P_n\}$, and; c) the procedure variables X_i in a definition $X_1(\tilde{x}_1) = P_1 \ \ldots \ X_n(\tilde{x}_n) = P_n$ are pairwise distinct.

2.2 Operational Semantics

Scope restriction and abstraction are the *variable binding operators* in the calculus. Variables $\tilde{x}$ are bound in the part P of a method $l(\tilde{x}) = P$, a procedure $X(\tilde{x}) = P$ or a scope restriction **new** $\tilde{x}$ P. The sets $\mathrm{fv}(P)$ and $\mathrm{bv}(P)$ of, respectively, *free variables* and *bound variables* of a process P are defined accordingly. Contrary to the conventional practice in process calculi, we let the scope of a **new** extend as far to the right as possible. A process P is said to be *closed for variables* if $\mathrm{fv}(P) = \emptyset$. We assume the usual definition of *simultaneous* substitution of values $\tilde{v}$ for the free occurrences of variables $\tilde{x}$ in a process P, $P\{\tilde{v}/\tilde{x}\}$. The substitution is defined only if the lengths of $\tilde{x}$ and $\tilde{v}$ match.

The *procedure variable binding operator* is the **def**. A procedure variable X is bound to Q in the process **def** $X(\tilde{x}) = P$ D **in** Q. The sets $\mathrm{ft}(P)$ and $\mathrm{bt}(P)$ of, respectively, *free procedure variables* and *bound procedure variables* of a process P are defined accordingly. A process P is said to be *closed for procedure variables* if $\mathrm{ft}(P) = \emptyset$. *Alpha conversion*, both for variables and procedure variables, is defined in the usual way.

Following Milner [18] we divide the computational rules of the language in two parts: the *structural congruence* rules and the *reduction* rules. Structural congruence rules allow the re-writing of processes into semantically equivalent expressions until they may be reduced using reduction rules.

We define $\equiv$ to be the smallest congruence relation over processes induced by the following rules.

$$
\begin{array}{rl}
[\textsc{Alpha}] & P \equiv Q \quad \text{if } P \equiv_\alpha Q \\
[\textsc{Group}] & P \mid \textsf{inaction} \equiv P \\
& P \mid Q \equiv Q \mid P, \\
& P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
[\textsc{New}] & \textsf{new}\ x\ \textsf{inaction} \equiv \textsf{inaction}, \\
& \textsf{new}\ x\ \textsf{new}\ y\ P \equiv \textsf{new}\ y\ \textsf{new}\ x\ P, \\
& \textsf{new}\ x\ P \mid Q \equiv \textsf{new}\ x\ (P \mid Q) \quad \text{if } x \notin \mathrm{fv}(Q) \\
[\textsc{Def}] & \textsf{def}\ D\ \textsf{in inaction} \equiv \textsf{inaction}, \\
& \textsf{def}\ D\ \textsf{in new}\ x\ P \equiv \textsf{new}\ x\ \textsf{def}\ D\ \textsf{in}\ P \quad \text{if } x \notin \mathrm{fv}\, D \\
& (\textsf{def}\ D\ \textsf{in}\ P) \mid Q \equiv \textsf{def}\ D\ \textsf{in}\ (P \mid Q) \quad \text{if } \mathrm{ft}(D) \cap \mathrm{ft}(Q) = \emptyset \\
& \textsf{def}\ D\ \textsf{in def}\ D'\ \textsf{in}\ P \equiv \textsf{def}\ D\ \textsf{and}\ D'\ \textsf{in}\ P \quad \text{if } \mathrm{dom}(D) \cap \mathrm{ft}(D') = \emptyset \\
[\textsc{Perm}] & \{l_1(\tilde{x}_1) = P_1 \quad l_2(\tilde{x}_2) = P_2\} \equiv \{l_2(\tilde{x}_2) = P_2 \quad l_1(\tilde{x}_1) = P_1\} \\
& X_1(\tilde{x}_1) = P_1 \quad X_2(\tilde{x}_2) = P_2 \equiv X_2(\tilde{x}_2) = P_2 \quad X_1(\tilde{x}_1) = P_1
\end{array}
$$

Rule Alpha ensures the equivalence of processes under *alpha conversion*. The rules in Group respectively garbage collect **inaction** processes, and state the commutativity and distributivity of the operator "|". Next, in New, the first rule garbage collects variables with empty scopes, the second commutes **new** statements and the third allows the scope of a newly introduced variable to be extended further to the right provided that there is no capture of variables. The rules in Def deal with definitions. The first garbage collects definitions with empty continuations. The second allows **new** statements to be moved in and out of **def** statements provided that there is no variable capture in the bodies of the procedures in D. The third rule allows the scope of a procedure variable to be extended as much to the right as possible, that is, without variable capture. The fourth allows procedure definitions to coalesce and be pushed to the start of a process. Finally, the rules in Perm, state that permutations of both method collections and procedure definitions are equivalent.

Reduction rules drive the computation either by making method calls, consuming pairs object-message, or by making procedure calls. Reduction does not necessarily contract a term. For example, a process of the form X with $\textsf{def}\ X = X \mid X$, expands to the process $X \mid X$ after one reduction step.

A message $x!l_i[\tilde{e}]$ selects the method l_i in an object $x?\{l_1(\tilde{x}_1) = P_1\ \ldots\ l_n(\tilde{x}_n) = P_n\}$. The result is the process P_i where the values $\tilde{v}$ resulting from the evaluation of the expressions in $\tilde{e}$ are bound to the variables in $\tilde{x}_i$. This form of reduction is called *method call*.

$$[\textsc{MethCall}] \qquad \frac{\tilde{e} \to \tilde{v}}{x\,!\,l_i[\tilde{e}] \mid x?\{l_1(\tilde{x_1}) = P_1\ \ldots\ l_n(\tilde{x_n}) = P_n\} \to P_i\{\tilde{v}/\tilde{x}_i\}}$$

Another form of reduction occurs when we call a procedure $X[\tilde{e}]$, defined elsewhere in the program as $X(\tilde{x}) = P$. The result is the process $P\{\tilde{v}/\tilde{x}\}$ where the values, $\tilde{v}$, of the expressions $\tilde{e}$ replace the variables $\tilde{x}$. This form of reduction is called *procedure call*.

$$[\textsc{ProcCall}] \qquad \frac{\tilde{e} \to \tilde{v}}{\textsf{def}\ X(\tilde{x}) = P\ D\ \textsf{in}\ X[\tilde{e}] \mid Q \to \textsf{def}\ X(\tilde{x}) = P\ D\ \textsf{in}\ P\{\tilde{v}/\tilde{x}\} \mid Q}$$

Finally, we have rules that allow reduction over concurrent composition, **new**s and **def**s, as well as the usual closure rule that allows processes to be re-written forming redexes:

$$[\textsc{Conc}] \quad \frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad [\textsc{New}] \quad \frac{P \to P'}{\textsf{new}\ x\ P \to \textsf{new}\ x\ P'}$$

$$[\textsc{Def}] \quad \frac{P \to P'}{\textsf{def}\ D\ \textsf{in}\ P \to \textsf{def}\ D\ \textsf{in}\ P'} \qquad [\textsc{Str}] \quad \frac{P' \equiv P \quad P \to Q \quad Q \equiv Q'}{P' \to Q'}$$

We define *multi-step reduction*, or simply *reduction*, notation $P \rightarrow^* Q$, as the relation $\equiv \cup \rightarrow^+$, where $\rightarrow^+$ is the transitive closure of $\rightarrow$.

Finally, we single out a channel – **io** – to be used for input/output. We assume a persistent object present at this channel that services all the IO requests from processes. The signature of this object is of the form:

$$\textbf{io}?\{\textbf{put}(x) = \ldots \ \textbf{get}(x) = \ldots\}$$

A message **io**!**put**["hello world"] sends the string "hello world" to the stdout, whereas a message **new** x **io**!**get**$[x]$ gets a value from the stdin and places it in a *val* labeled message to x.

This completes the operational semantics of the language.

2.3 Derived Constructs

We introduce a few derived constructs in TyCO to ease the programming task by making the processes more readable. Some of these constructs are so widespread in TyCO programs that they are also optimized internally by the compiler. The builtin conditional construct and its single branch form are standard constructs in concurrent programming. We single out a label, *val*, to be used in objects with a single method. This allows the programmer to abbreviate messages and objects.

$$\begin{array}{lcl} \textbf{if } e \textbf{ then } P & \equiv & \textbf{if } e \textbf{ then } P \textbf{ else inaction} \\ x![\tilde{e}] & \equiv & x!\mathit{val}[\tilde{e}] \\ x?(\tilde{y}) = P & \equiv & x?\{\mathit{val}(\tilde{y}) = P\} \end{array}$$

The construct **val** is used to bind a variable to the value of an expression in the context of a process P. No communication is involved. The ";" operator is the sequential composition operator. Here it is defined for synchronous method calls and synchronous procedure calls.

$$\begin{array}{lcl} \textbf{val } \tilde{x} = \tilde{e} \textbf{ in } P & \equiv & ((\tilde{x})P)[\tilde{e}] \\ x!l[\tilde{e}] \ ; \ P & \equiv & \textbf{new } y \ x!l[\tilde{e}y] \mid y?() = P \\ X[\tilde{e}] \ ; \ P & \equiv & \textbf{new } y \ X[\tilde{e}y] \mid y?() = P \end{array}$$

We also define two constructors for two special and particularly pervasive cases of synchronization: synchronous procedure call and pattern matching. The **let** constructor is quite useful in getting back results from synchronous calls; the syntax is taken from Pict [6]. Finally, the **match** constructor is most useful for pattern matching; the syntax is taken from ML [20].

$$\begin{array}{lcl} \textbf{let } y = x!l[\tilde{e}] \textbf{ in } P & \equiv & \textbf{new } z \ x!l[\tilde{e}z] \mid z?(y) = P \\ \textbf{let } y = X[\tilde{e}] \textbf{ in } P & \equiv & \textbf{new } z \ X[\tilde{e}z] \mid z?(y) = P \\ \textbf{match } x!l[\tilde{e}] \textbf{ with } M & \equiv & \textbf{new } y \ x!l[\tilde{e}y] \mid y?M \\ \textbf{match } X[\tilde{e}] \textbf{ with } M & \equiv & \textbf{new } y \ X[\tilde{e}y] \mid y?M \end{array}$$

2.4 Programming Examples

In this section we introduce two small programming examples in TyCO to allow the reader to be more familiar with the programming style and semantics of the language.

A Memory Cell is parameterized by its address in memory, address, and the value it holds (listing 1). When an instance of such a memory cell is created, Cell[x,2], an object with two methods, read and write, is deployed at address x in memory.

Listing 1: A Memory Cell

```
def MemoryCell(address, value) =
```

```
    address ? {
        {-- read from cell --}
        read(replyTo) =
                replyTo![value] |
                MemoryCell[address, value]

        {-- write to cell --}
        write(newValue) =
                MemoryCell[address, newValue]
    }
in
   {-- .... create instances and use them here ... --}
   new x MemoryCell[x,2] | let y = x!read[] in io!puti[y]
```

The type of elements the cell at address x will hold will always be int. Because of the language type system, each new address can hold values of exactly one type only. Thus we could create another memory cell at a new address for holding booleans, but never at x. The example creates a memory cell with an integer 2, reads its value into variable y and prints it.

The execution of the above program can be seen using the following reduction, where we let $C(a\ v)$ denote the definition of MemoryCell to make the expressions more compact.

$$
\begin{aligned}
&\textbf{def}\ C(a\ v) = \ldots\ \textbf{in new}\ a_1\ C[a_1\ 2] \mid \textbf{let}\ y = a_1!read[]\ \textbf{in io}!puti[y] \\
&\equiv && \text{(by definition)} \\
&\textbf{def}\ C(a\ v) = \ldots\ \textbf{in new}\ a_1\ C[a_1\ 2] \mid \textbf{new}\ r\ (a_1!read[r]\ \mid r?\{val(y) = \textbf{io}!puti[y]\}) \\
&\equiv && \text{(New)} \\
&\textbf{def}\ C(a\ v) = \ldots\ \textbf{in new}\ a_1\ r\ C[a_1\ 2] \mid a_1!read[r]\ \mid r?\{val(y) = \textbf{io}!puti[y]\} \\
&\equiv && \text{(Def)} \\
&\textbf{new}\ a_1\ r\ \textbf{def}\ C(a\ v) = \ldots\ \textbf{in}\ C[a_1\ 2] \mid a_1!read[r]\ \mid r?\{val(y) = \textbf{io}!puti[y]\} \\
&\equiv && \text{(ProcCall)} \\
&\textbf{new}\ a_1\ r\ \textbf{def}\ C(a\ v) = \ldots\ \textbf{in}\ a_1?\{\ldots\} \mid a_1!read[r]\ \mid r?\{val(y) = \textbf{io}!puti[y]\} \\
&\equiv && \text{(MethCall)} \\
&\textbf{new}\ a_1\ r\ \textbf{def}\ C(a\ v) = \ldots\ \textbf{in}\ r!val[2] \mid C[a_1\ 2] \mid r?\{val(y) = \textbf{io}!puti[y]\} \\
&\equiv && \text{(Group,MethCall)} \\
&\textbf{new}\ a_1\ r\ \textbf{def}\ C(a\ v) = \ldots\ \textbf{in}\ C[a_1\ 2] \mid \textbf{io}!puti[2]
\end{aligned}
$$

Thus, the value in the memory cell is read and written to the I/O system.

A List Values is a datatype that can be defined by two process definitions: cons and nil. A Cons process is an object that responds to val method calls. The call is made with a replyTo and the method sends a message labeled cons with the head and the tail of the list as arguments to whoever is listening on replyTo. The Nil process is a similar object but simply sends an empty message with label nil (listing 2).

Listing 2: A List

```
def Nil(address) =
        address ? val(replyTo) =
            {-- send nothing --}
            replyTo!nil[] |
            Nil[address]

    Cons(address, head, tail) =
        address ? val(replyTo) =
            {-- send head and tail --}
            replyTo!cons[head, tail] |
            Cons[address, head, tail]

    ReadList(address) =
```

```
        match address with {
            nil() =
                inaction
            cons(head, tail) =
                io!puti[head] |
                ReadList[tail]
in
    {-- .... a list of integers ... --}
    new x, y, z Nil[x] | Cons[y,1,x] | Cons[z,2,y] | ReadList[z]
```

The process ReadList takes an address for a list and iterates through it, printing all the values to the I/O system as it goes.

Again, we can observe the execution of the above program using the following reduction, where we let $N(a)$ denote the definition of Nil, $C(a\ h\ t)$ the definition of Cons and, $R(a)$ the definition of ReadList.

$$
\begin{aligned}
&\textbf{def}\ N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots\ \textbf{in} \\
&\textbf{new}\ a_1\ a_2\ a_3\ N[a_1] \mid C[a_2\ 1\ a_1] \mid C[a_3\ 2\ a_2] \mid R[a_3] \\
&\equiv && \text{(Def)} \\
&\textbf{new}\ a_1\ a_2\ a_3\ \textbf{def}\ N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots\ \textbf{in} \\
&N[a_1] \mid C[a_2\ 1\ a_1] \mid C[a_3\ 2\ a_2] \mid R[a_3] \\
&\equiv\rightarrow && \text{(ProcCall)} \\
&\textbf{new}\ a_1\ a_2\ a_3\ \textbf{def}\ N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots\ \textbf{in} \\
&a_1?val(r) = r\,!\,nil[] \mid N[a_1] \mid \\
&C[a_2\ 1\ a_1] \mid C[a_3\ 2\ a_2] \mid R[a_3] \\
&\equiv\rightarrow && \text{(Group,ProcCall)} \\
&\textbf{new}\ a_1\ a_2\ a_3\ \textbf{def}\ N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots\ \textbf{in} \\
&a_1?val(r) = r\,!\,nil[] \mid N[a_1] \mid \\
&a_2?val(r) = r\,!\,cons[1\ a_1] \mid C[a_2\ 1\ a_1] \mid C[a_3\ 2\ a_2] \mid R[a_3] \\
&\equiv\rightarrow && \text{(Group,ProcCall)} \\
&\textbf{new}\ a_1\ a_2\ a_3\ \textbf{def}\ N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots\ \textbf{in} \\
&a_1?val(r) = r\,!\,nil[] \mid N[a_1] \mid \\
&a_2?val(r) = r\,!\,cons[1\ a_1] \mid C[a_2\ 1\ a_1] \mid \\
&a_3?val(r) = r\,!\,cons[2\ a_2] \mid C[a_3\ 2\ a_2] \mid R[a_3] \\
&\equiv\rightarrow && \text{(Group,ProcCall)} \\
&\textbf{new}\ a_1\ a_2\ a_3\ \textbf{def}\ N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots\ \textbf{in} \\
&a_1?val(r) = r\,!\,nil[] \mid N[a_1] \mid \\
&a_2?val(r) = r\,!\,cons[1\ a_1] \mid C[a_2\ 1\ a_1] \mid \\
&a_3?val(r) = r\,!\,cons[2\ a_2] \mid C[a_3\ 2\ a_2] \mid \\
&\textbf{match}\ a_3\ \textbf{with}\ldots \\
&\equiv
\end{aligned}
$$

The ReadList process now starts to iterate through the list and generating printing processes for all the elements of the list:

$\equiv$ (by definition)

new $a_1\ a_2\ a_3$ **def** $N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots$ **in**
$a_1?val(r) = r\,!\,nil[]\ |\ N[a_1]\ |$
$a_2?val(r) = r\,!\,cons[1\ a_1]\ |\ C[a_2\ 1\ a_1]\ |$
$a_3?val(r) = r\,!\,cons[2\ a_2]\ |\ C[a_3\ 2\ a_2]\ |$
new $r\ a_3\,!\,val[r]\ |\ r?\{nil() = \ldots\ cons(h\ t) = \ldots\}$

$\equiv$ (New)

new $a_1\ a_2\ a_3$ **def** $N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots$ **in**
$a_1?val(r) = r\,!\,nil[]\ |\ N[a_1]\ |$
$a_2?val(r) = r\,!\,cons[1\ a_1]\ |\ C[a_2\ 1\ a_1]\ |$
new $r\ a_3?val(r) = r\,!\,cons[2\ a_2]\ |\ C[a_3\ 2\ a_2]\ |$
$a_3\,!\,val[r]\ |\ r?\{nil() = \ldots\ cons(h\ t) = \ldots\}$

$\equiv\!\rightarrow$ (New,MethCall)

new $a_1\ a_2\ a_3$ **def** $N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots$ **in**
$a_1?val(r) = r\,!\,nil[]\ |\ N[a_1]\ |$
$a_2?val(r) = r\,!\,cons[1\ a_1]\ |\ C[a_2\ 1\ a_1]\ |$
new $r\ C[a_3\ 2\ a_2]\ |\ r\,!\,cons[2\ a_2]\ |\ r?\{nil() = \ldots\ cons(h\ t) = \ldots\}$

$\rightarrow$ (MethCall)

new $a_1\ a_2\ a_3$ **def** $N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots$ **in**
$a_1?val(r) = r\,!\,nil[]\ |\ N[a_1]\ |$
$a_2?val(r) = r\,!\,cons[1\ a_1]\ |\ C[a_2\ 1\ a_1]\ |$
new $r\ C[a_3\ 2\ a_2]\ |$ **io**$\,!\,puti[2]\ |\ R[a_2]$
$\ldots$
$\equiv^*\!\rightarrow^*$
$\ldots$
new $a_1\ a_2\ a_3$ **def** $N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots$ **in**
new$r\ N[a_1]\ |\ Cons[a_2 1\ a_1]\ |\ C[a_3\ 2\ a_2]\ |$
io$\,!\,puti[2]\ |$ **io**$\,!\,puti[1]\ |$ **inaction**

$\equiv^*$ (New, Group)

new $a_1\ a_2\ a_3$ **def** $N(a) = \ldots\ C(a\ h\ t) = \ldots\ R(a) = \ldots$ **in**
$N[a_1]\ |\ Cons[a_2 1\ a_1]\ |\ C[a_3\ 2\ a_2]\ |$
io$\,!\,puti[2]\ |$ **io**$\,!\,puti[1]$

$\equiv^*$ (New, Group)

3 The Compiler and Run-Time

The compiler for the TyCO programming language is divided into three basic steps: syntactic analysis, semantic analysis and code generation (figure 1). The syntactic analysis is performed by a parser generated using the tools JFlex [2] and JCup [13]. At the end of this phase an abstract syntax tree is built for the program and it is feed into the next compilation stage. The semantic analysis performs three operations on the syntax tree: variable scope resolution, space allocation and type inference. These operations annotate the syntax tree with information that is required by the code generation phase. Well typed TyCO programs pass successfully by the type-inference system and are hence-

forward guaranteed to be devoid of run-time protocol errors (e.g., calling a non-existing method in an object, or a non-existing procedure call, or using the wrong type or wrong number of arguments in calls).

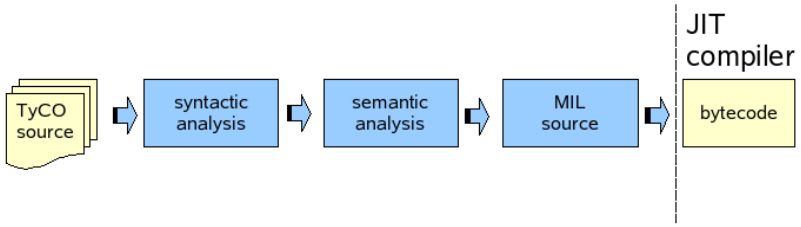

Figure 1: The compilation of a TyCO program

The final compilation stage is code generation. Here, the annotated syntax tree is processed and the resulting program is generated in an intermediate language called MIL (Multithreaded Intermediate Language).

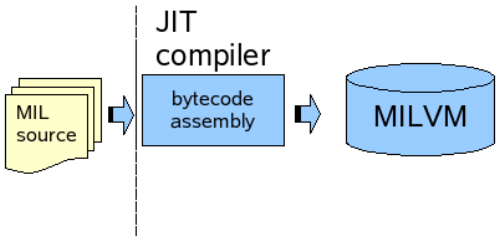

Figure 2: The execution of a compiled TyCO program

The MIL programs generated by the compiler may be run with the TyCO run-time system, which implements the MIL instruction set architecture (figure 2). Before running a MIL program, however, the run-time system performs a preliminary JIT (Just In Time) compilation of the program into an internal byte-code format. This extra step is included in the TyCO run-time since it allows a more flexible manipulation of the MIL program. It can be used to perform dynamic type-checking in a distributed version of the system (already available) or to implement dynamic linking in general.

3.1 The Target Language

MIL is a multi-threaded (typed) assembly language [16, 26] designed to be the target of the compilation of process-calculi based languages. A MIL program is composed of an arbitrary number of **code** and **data** fragments. **code** fragments hold the code for activation records that will be created at run-time. **data** fragments, on the other hand, contain static data such as method tables and constants. Programs must also include two directives, **version** and **registers**, that indicate, respectively, the version of the MIL run-time and the number of general purpose registers it requires to run.

The syntax of MIL programs is given by the following grammar:

P	::=	**version** $i.i$ **registers** i C $\tilde{F}$	Program
F	::=	$C \mid D$	Fragments
C	::=	**code** l $\{\tilde{I}$ **schedule**$\}$	Code fragment
D	::=	**data** l $\{\tilde{w}\}$	Data fragment
I	::=		Instructions
		$r :=$ **malloc** i	Allocate heap space
	$\mid$	**launch** r	Launch new activation record
	$\mid$	**external** c r	External service call
	$\mid$	$r := v \mid r := v[v] \mid r[v] := v$	Moving data
	$\mid$	**if** r bop v **jump** v	Conditional jump
	$\mid$	**jump** v	Unconditional jump
	$\mid$	$r := r$ aop v	Arithmetic operations
	$\mid$	r.**lock** ()	
	$\mid$	r.**unlock** ()	Exclusive access
r	::=	$r0 \mid r1 \mid r2 \mid \ldots$	Registers
w	::=	$l \mid i$	Word values
v	::=	$w \mid r$	General values
aop	::=	$+ \mid - \mid * \mid / \mid \%$	Arithmetic operations
bop	::=	$== \mid != \mid > \mid < \mid >= \mid$	
		$<= \mid @== \mid @!=$	Branch operations

Word values w comprise integers and heap labels. General values incorporate, in addition, registers. In other words, general values may contain registers whereas word values may not. External service identifiers in MIL are integer constants c. Labels, l, are sequences of letters, digits, hashes #, or _, starting with a letter. Constants, c, denote external services. Registers are represented by ri, where i is non-negative integer. At run-time, the register $r0$ always keeps the reference for the current activation record.

The concrete syntax of the language imposes some syntactic restrictions to the above grammar, such as: (a) the number of a register must be within the interval with boundaries 0 and the number defined in the **registers** directive minus 1; (b) labels are unique, and cannot be re-declared, and; (c) the target of a jump instruction must be defined within the code of the current **code** fragment.

Besides the usual assignment, conditional execution, arithmetic and relational operations, the remainder of the MIL instructions may be divided into sets, according to the type of operations they perform and the data-structures they manipulate.

Memory allocation. Instruction **malloc** allocates a fresh, uninitialized, *frame* in the heap with given size, and returns a reference for it. Frames can hold channels, objects or messages targeted at a channel, activation records for **code** fragments, or the arguments of an external service (figure 3).

Frames that hold channels have a special treatment that will be discussed in detail below. Frames that hold messages are placed in a queue associated with the target channel, and have the following form: $\langle$next, methodName, arguments, _$\rangle$, where next points to the next element in the queue, methodName is the label of the message, and arguments points to a frame that holds the arguments of the method (figure 3). Frames that hold objects are also placed in the queue of a channel, and have the following form: $\langle$next, methodName, _, freeVars$\rangle$, where freeVars points to a frame that holds the variables that occur free in the bodies of its methods (figure 3). Activation records for **code** fragments are queued in the virtual machine's run-queue and their form has, in addition to the next field, a *code address* field that points to the address where the code for the **code** fragment is: $\langle$next, code, arguments, freeVars$\rangle$.

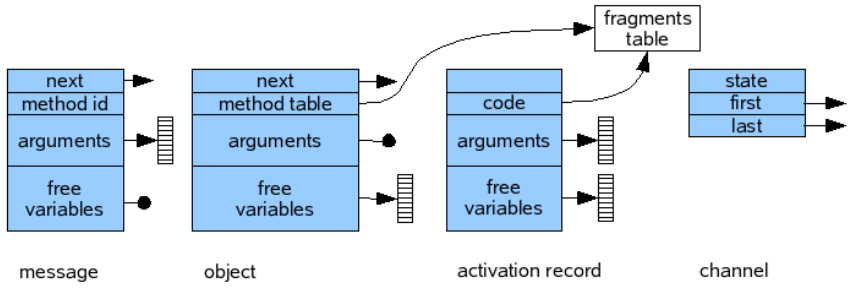

Figure 3: Types of frames in the MIL VM

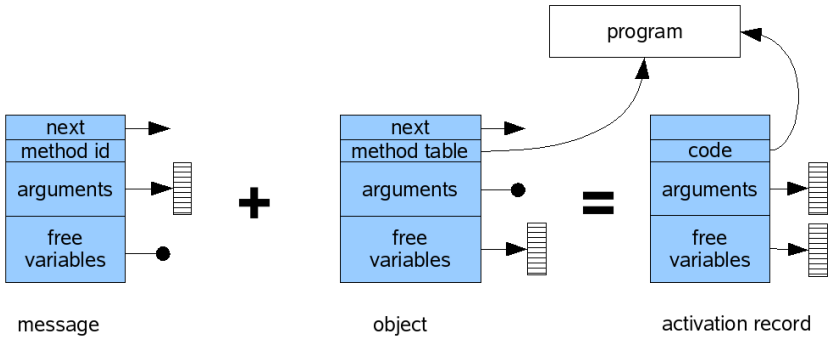

Figure 4: Frames involved in a method call

An activation record is created whenever a method call (figure 4) is performed successfully. In this case the frame of the object, holding the reference for the method table and the free variables in the object, is combined with the frame of the message holding the method label and the arguments to the call, to get the activation record. This corresponds to the reduction rule MethCall in the operational semantics of the language:

$$\frac{\tilde{e} \rightarrow \tilde{v}}{x\,!\,l_i[\tilde{e}] \mid x?\{l_1(\tilde{x}_1) = P_1 \ \ldots \ l_n(\tilde{x}_n) = P_n\} \rightarrow P_i\{\tilde{v}/\tilde{x}_i\}}$$

Another possible way to create an activation record is through a procedure call (figure 5). In this case, the frame corresponding to the procedure and holding the code reference and the free variables in the code is combined with the frame from the call, that holds the arguments of the call, to form the final record.

This corresponds to the reduction rule ProcCall in the operational semantics of the language:

$$\frac{\tilde{e} \rightarrow \tilde{v}}{\textbf{def}\ X(\tilde{x}) = P\ D\ \textbf{in}\ X[\tilde{e}] \mid Q \rightarrow \textbf{def}\ X(\tilde{x}) = P\ D\ \textbf{in}\ P\{\tilde{v}/\tilde{x}\} \mid Q}$$

Finally, a frame with arguments for an external service is simply a frame of words: $\langle w_1, \ldots\ w_{n-1}\rangle$. If the operation returns a result, it is usually placed in the first slot of the frame. For instance, the frame to call a service that implements a function, such as x^n, is $\langle x, n\rangle$ (figure 9). Its contents after the service execution will be: $\langle result, \_\rangle$.

Exclusive access to the heap frames is required in a multithreaded implementation of the virtual machine. Thus, we provide two basic instructions to support locks. r.**lock** provides exclusive access to the frame held in r to the calling thread, if it succeeds. Otherwise it suspends the current thread.

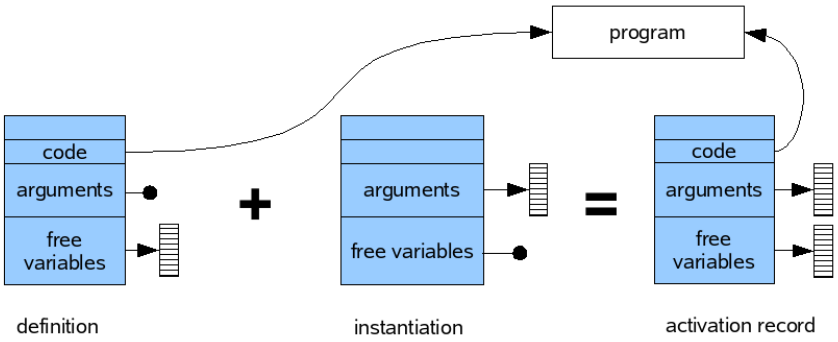

Figure 5: Frames involved in a procedure call

r.**unlock** releases the lock so that other threads may gain access to the frame held in r, if the current thread owns the lock. It does nothing otherwise.

Activation record manipulation. Instruction **launch** enqueues a frame holding an activation record in the run-queue of the virtual machine. Instruction **schedule** terminates the execution of the current activation record and searches the run-queue for more work.

External service call. Instruction **external** allows the execution of calls to code outside the MIL virtual machine. It receives as input a constant (the service identifier) and a reference for the frame with the arguments of the call. The constant denoting the service is composed by two identifiers, the service *module* and, the name of the *method* in the module, e.g., *string.concat* for the *concat* method in module *string*.

This facility of the MIL virtual machine will allow us to implement the debugger for the language in a seamless way.

Derived constructs to manipulate channels: Channels are special frames in the heap with three slots. The slots hold the first and last elements of a queue and the composition of the queue, as illustrated in figure 6. Channels in TyCO obey to the invariant that, at one given time, they can only be empty, or have only objects pending or have only messages pending.

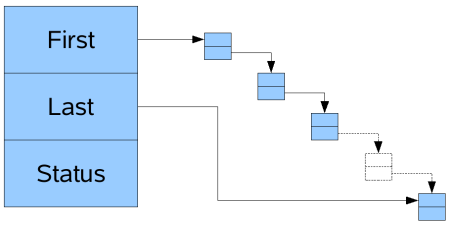

Figure 6: A channel frame

To create and manipulate channels we define a set of macros:

- r := **newQueue** (): creates and initializes a frame to hold a channel, and returns a reference to it;
- r.**dequeue** (): removes a frame from the queue referenced by r, updates the state of the queue, and returns a reference for the frame;
- r.**enqueueObj** () and r.**enqueueMsg** () place a frame in the queue referenced by register r and update the state of the queue;

- r.**getStatus** () returns 0 if the queue is empty; a positive integer if there are objects pending in the queue, or; a negative integer if there are messages pending in the queue. The modulus of the integer value is the number of objects, respectively messages, in the queue.

As an example we provide the definition of the **enqueueMsg** macro. To ease the reading, we refer to a given position within a frame by the name associated to its contents (e.g., next, status). Assume that the register holding the frame for the channel is $r1$ and the frame to be enqueued is in register $r2$.

$$
\begin{array}{lcl}
r1.\textbf{enqueueMsg}\ (r2) & \stackrel{\text{def}}{=} & \begin{array}{l}
\quad r3 = r1[\mathsf{status}] \\
\quad \textbf{if}\ r3 == 0\ \textbf{jump}\ \mathsf{empty} \\
\quad r3 := r1[\mathsf{last}] \\
\quad r3[\mathsf{next}] := r2 \\
\quad r1[\mathsf{last}] := r2 \\
\quad r3 = r1[\mathsf{status}] \\
\quad r4 = r3 + 1 \\
\quad r1[\mathsf{status}] = r4 \\
\quad \textbf{jump}\ \mathsf{done} \\
\mathsf{empty}: \\
\quad r1[\mathsf{first}] := r2 \\
\quad r1[\mathsf{last}] := r2 \\
\quad r1[\mathsf{status}] = 1 \\
\mathsf{done}:
\end{array}
\end{array}
$$

3.2 The Run-Time System

The semantics of the MIL language is defined as an abstract transition machine [16, 26]. A state of the machine (figure 7) is composed of:

- a *Run-queue*, denoted by Q, is a queue of ready-to-execute MIL activation records, located in the heap;
- a *Heap*, denoted by H, is the heap of the machine containing heap values that might be frames or code fragments;
- a set of n *Registers*, denoted by R, where n is the number given in the **registers** directive;
- a *Sequence of Fragments*, denoted by I, is the source code of the program to be executed.

$$
\begin{array}{llll}
\textit{Small Values} & v & ::= & w \mid r \mid ? \\
\textit{Run-Queue} & Q & ::= & \textit{nil} \mid l :: Q \\
\textit{Heap Values} & h & ::= & \langle w_1, \ldots, w_n \rangle \mid I \\
\textit{Heaps} & H & ::= & \langle l_1 \mapsto h_1, \ldots, l_n \mapsto h_n \rangle \\
\textit{Registers} & R & ::= & \langle r_0 \mapsto w_0, \ldots, r_{n-1} \mapsto w_{n-1} \rangle \\
\textit{State} & P & ::= & \langle Q, H, R, I \rangle \mid \textit{halt}
\end{array}
$$

Figure 7: States of the MIL machine

Small values are extended with the notion of uninitialised value ?, to denote values that have not yet be initialized. A program,

$$\textbf{version}\ n.n\ \textbf{registers}\ n\ \textbf{code}\ l_0\{I_0\}\ \mathsf{frag}\ l_1\{h_1\} \ldots \mathsf{frag}\ l_k\{h_k\}$$

is loaded to the machine through a load macro defined as follows:

$$\mathsf{load}\ (\textbf{version}\ n.n\ \textbf{registers}\ n\ \textbf{code}\ l_0\{I_0\}\ \mathsf{frag}\ l_1\{h_1\}\ldots\mathsf{frag}\ l_k\{h_k\}) = \langle nil, \langle l_1 \mapsto h_1, \ldots, l_k \mapsto h_k\rangle, \langle r_0 \mapsto ?, \ldots, r_n \mapsto ?\rangle, I_0\rangle$$

where each frag represents one of the keywords **code** or **data**. The machine terminates its execution when it reaches the *halt* state. The reduction rules are of the form:

$$\langle Q, H, R, I\rangle \rightarrow \langle Q', H', R', I'\rangle$$

with one rule per MIL instruction. We refer the reader to [26] for the details of the virtual machine.

The run-time system for TyCO is an implementation of the MIL virtual machine on top of the Java Virtual Machine (figure 8). The implementation is straightforward. The set of registers is implemented as an array, the run-queue as a queue of activation record objects and, the program as a hash table mapping labels onto program fragments. The heap is managed by the JVM.

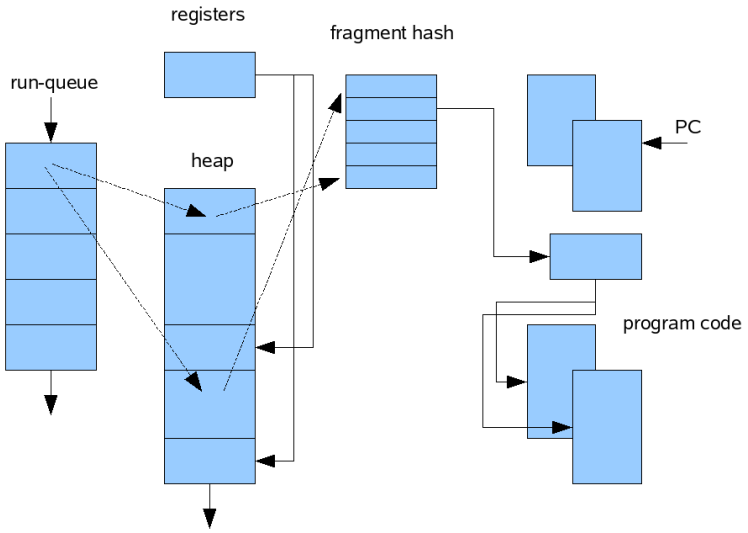

Figure 8: The architecture of the run-time system

When a .mil file is passed to the run-time, it is first compiled into a byte code representation, very close to MIL itself, and then passed to the run-time proper. The run-time engine is implemented in a rather simple way through a class VirtualMachine that extends the Java Thread class and implements an interface InstructionSet. The core of the action lies in the main loop that consists of a switch-case construct with one case per MIL instruction (listing 3).

Once the program code is transformed into byte-code, it is added to the fragment table and the first activation record, corresponding to the `main` fragment, is placed in the run-queue. This activation record is picked up from the run-queue and the instructions in the fragment are executed. Instructions within each fragment are executed sequentially within the engine loop. Each time a successful method call or procedure call occurs, a new activation record is generated and placed in the run-queue, ready to be executed. The instructions schedule and launch are used to get new activation records from the run-queue and to dispatch new activation records to it, respectively.

The run-time halts when the code for the current activation record ends and there are no more activation-records available in the run-queue. This condition is sufficient for a single threaded implementation of the virtual machine.

Listing 3: The Run-Time Engine

```
public class VirtualMachine extends Thread implements InstructionSet {

  // atributes
  protected PrivateRunQueue runQueue;
  protected int pc;
  protected org.tyco.mil.vm.values.Value[] registers;
  protected int[] program;
  protected VirtualMachine machine;

  // constructor
  ...

  // run()
  public void run() {
    try {
      // get main activation record from the run-queue
      schedule();
      // run the program.
      exec();
    }
    // catch exceptions
    ...
  }

  protected void exec() throws InterruptedException {
    while (this.machine.isRunning()) {
      switch (this.program[this.pc]) {
        ...

        case org.tyco.mil.vm.assemble.OpCodes.SCHEDULE :
        schedule();
        break;

        ...

        default:
        // throw exception
      }
    }
  }

  // Methods of the InstructionSet interface

  ...

  public void schedule() throws InterruptedException {
    org.tyco.mil.vm.values.Frame frame = this.runQueue.dequeue();
    // switch to the new thread
    if (frame != null)
      switchContext(frame);
  }
  ...

}
```

In the current multithreaded implementation of TyCO (not yet supported by the debugger), each thread has a private run-queue to put to and get from activation records. Naturally, a simple check of the local run-queue is not enough. In addition to these local run-queues, the threads also access a global run-queue where they may again put, or get, activation-records. The multi-threaded virtual machine resorts to this run-queue to halt the threads. When the private run-queue of a thread is empty it tries to get work from the global run-queue. If no work exists in this queue, them the thread suspends on the queue. If, at some instant, all threads are suspended on the global run-queue, the machine halts [22].

4 The Debugger

In this section we describe how we have implemented a debugger for the TyCO programming language by making adequate interventions in the code for the compiler and run-time system. Our goal was to provide a user-level debugger for TyCO users. Such a tool provides fundamental support for users

in developing their applications in the unique programming paradigm of TyCO, without exposing the virtual machine internals. Thus, users may follow the execution of their programs and view the basic abstractions of the language such as channels, messages, objects, procedures and, method and procedure calls as they evolve at run-time.

The debugger here described was implemented in a modular way relative to the compiler and virtual machine. The instructions inserted by the compiler in the final MIL code of a debug-enabled program are part of a debug module whose implementation is orthogonal to the compiler and the run-time system. The module is accessed from within the MIL code through external calls. In this fashion the functionalities of the debugger can be easily extended in future implementations.

4.1 Modifications to the Compiler

Since we are interested in a user-level debugger, our main goal is to be able to relate source code abstractions such as instructions and variables with run-time data-structures such as channels, frames and activation records.

For this purpose we changed the implementation of the TyCO compiler in order to preserve some relevant information from the parsing phase, namely, the lexemes for the identifiers and their positions in the source code (line and column).

In the semantic analysis, more specifically space allocation, we use this information to map source code identifiers onto machine data-structures such as registers, channels and frames. We keep this information in hash tables that are passed on to the next stage.

The code generation stage takes this information and instruments the final MIL code with extra instructions that basically write down the information in these tables to the final MIL code. The extra instructions are implemented as external calls to a *debug* module, and thus are included in a modular way in the virtual machine instruction set.

The calls to the *debug* module obey to the same layout as all external calls. The call takes two arguments, the first identifying the module and method to be invoked and the second a register that holds a reference for a frame with the arguments of the call (figure 9).

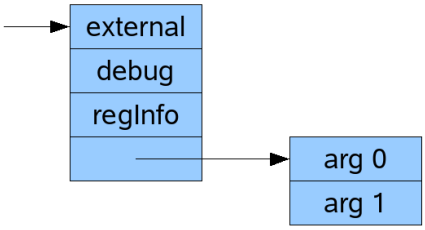

Figure 9: Frame for external call

We provide two examples of calls to the *debug* module. The first example, *debug.procInfo* calls a method *procInfo*. The MIL code for the call is:

external *debug.procInfo r*

When the call is executed, it outputs information relative to the instruction currently being executed such as the line and column numbers where it appears in the source code.

Another call, *regInfo*, written as:

external *debug.regInfo r*

outputs information relative to a given identifier in the source code such as the line and columns where it appears and the value it holds at the moment. To associate these debugging instructions with instructions and identifiers in a TyCO program, the compiler attaches one of these instructions to each variable binder appearing in the source code (in the case of *regInfo*) and to every instruction in the source code (in the case of *procInfo*). Take the following example of a **newQueue** instruction:

$r3 :=$ **newQueue**
$r4 :=$ **malloc** (2)
$r4[0] :=''$ x''
$r4[1] := pack(1, 5, 3)$
external *debug.regInfo* $r4$
$r4 :=$ **malloc** (2)
$r4[0] :=''$ New''
$r4[1] := pack(1, 5)$
external *debugger.procInfo* $r4$

The above code adds debugging information for the **newQueue** instruction by calling *procInfo* and, since this instruction is also a binder for channels in TyCO, it adds debugging information for the new variable being created.

First, it creates a new frame to hold the lexeme for the variable created ("x"), its line (1) and column (5) in the source code and, the number of the register (3) it is mapped to.

In some cases, the variables may not be represented directly in the source code. They may be anonymous variables generated by the compiler as a result, for example, of the compilation of derived constructs such as **let-in** or **match-with**. For the sake of completeness, the compiler also includes information for these variables although this option is somewhat contradictory to our goal of having a user-level debugger.

Second, the compiler creates another frame with information on the instruction being executed in the source code ("new") and the line (1) and column (5) where it happens. The *pack* macro simply packs the arguments into a machine word.

The debugging calls were inserted in the code generation phase in places where there may be some change in the state of the virtual machine and this has a direct correspondence with instructions in the TyCO source code. Here are a few of these situations: (a) creation of new channels; (b) creation of objects; (c) method invocations; (d) procedure calls, and; (e) changes in the state of a channel.

4.2 Modifications to the Run-Time

We now face the problem of controlling the execution of programs by the debugger. One possible solution would be to create a custom version of the virtual machine that included the control code and data-structures required to manage the execution of the debugger. However, this solution had two major inconvenient. First, this approach is not modular and we wanted to decouple as much as possible the virtual machine and the control and interaction code for the debugger. Second, while the solution could be admissible in a single threaded implementation of TyCO, it would be extremely cumbersome to use in the current multi-threaded implementation. An architecture with one thread for the virtual machine and an external thread to implement the control and interaction of the debugger is more scalable.

With these considerations we have implemented the TyCO debugger with two concurrent threads. One of the threads runs the virtual machine loaded with the *debug* module at the beginning of the execution. The other runs an interactive shell for the user and controls the execution of the program in the virtual machine (figure 10).

The synchronization between the two threads is achieved through a shared object, that halts one of

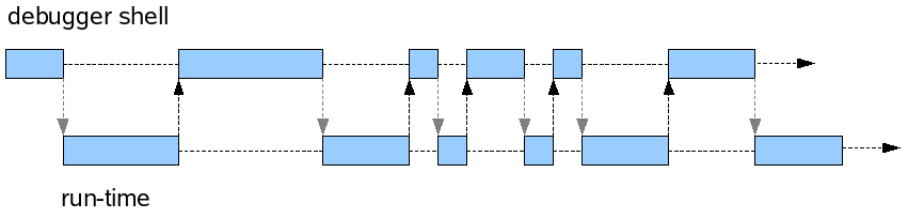

Figure 10: Synchronization between the debugger shell and run-time

the threads and restarts the other. This switching occurs in response to events in the execution of the program (on the side of the virtual machine) or in response to commands (on the side of the command shell).

More specifically, the virtual machine thread is suspended and the command shell is restarted whenever a breakpoint is defined for the line of the source code being executed or, whenever a variable that is being spied changes its state. The command shell, on the other hand, is suspended and the virtual machine restarted whenever one of the commands run, continue, next or step is executed.

Initially, when the debugger is started, only the command shell thread is running. This allows the user to define parameters such as the program arguments, a set of breakpoints or the location of the source code file. At the end of the execution of the debugger, control is given back to the command shell thread, to terminate correctly.

The data structures used by the command shell to control the execution of the program are rather simple. The most important ones are the breakpoint and spying tables. As the name implies these are hash tables used to keep the active breakpoints in a program and the variables being spied on. These data-structures are common to both the command shell and the virtual machine. The shell updates the tables when the user adds or removes breakpoints or variables to spy. The virtual machine consults the information on the tables in order to decide whether or not to yield the control back to the shell.

4.3 Functionality

The debugger has functionalities typical of the gdb debugger for control of the execution and for setting breakpoints. Assuming x is a variable and n an integer, the basic commands it implements are:

run	starts running the program
continue	continues the execution of the program
step	continues until the next operation on a channel
breakpoint n	adds a new breakpoint for line n to the table
remove n	removes the breakpoint at line n
spy x	adds a new breakpoint for variable x
nospy x	removes the breakpoint for variable x
list	prints the list of the breakpoints
exit	exits the debugger
help	prints this information

The main distinction from the standard gdb commands comes from the visualization of information. Here, the abstractions we are interested in observing are channels and frames corresponding to messages, objects or procedure calls. These are specific to TyCO and in general to languages based on process calculi.

print table	prints all variables mapped onto registers
print register x	prints the register mapped to variable x
print register n	prints the contents of the n-th register
print status x	prints the status of variable x
print queue x	prints the contents of variable x
print object n x	prints the n-th object in variable x
print message n x	prints the n-th object in variable x
print run-queue	prints the activation records
print record n	prints the n-th activation record

This small set of commands allows users to have a firm grip on the run-time evolution of a program.

4.4 Using the Debugger

To use the TyCO debugger we added a flag to the compiler of the language, -g, which signals the compiler to attach debugging information to the final MIL code (listing 4).

Listing 4: Compiling a TyCO Program with Debugging Information

```
> java org.tyco.tyco.tycoc -g program.tyco
```

The resulting program.mil file can then be executed in the TyCO run-time (listing 5).

Listing 5: Running the Debugger

```
> java org.tyco.tyco.tycodb -s ../program.tyco program.mil
```

The TyCO debugger assumes that the source file for program.mil resides in one of a configurable set of directories. An optional argument, -s (as above), may be used to indicate the location of the source file.

Listing 6 shows a small program in TyCO, program.tyco, which we use to demonstrate a few commands from the debugger. The program defines a process print that waits for messages on a channel x carrying integers. When it receives one such message it prints the integer and recursively listens for more messages.

Listing 6: Line Breakpoints in the TyCO Debugger

```
new x
def
    Print() =
      x ? (y) = io!printi[y] | Print[]
in
    Print[] | x ! [1] | x ! [2] | x ! [3]
```

After compiling this TyCO program we run the debugger on program.mil (listing 7). As usual, we may simply run the program without breakpoints with run. More commonly, tough, we want to inspect the run-time data-structures associated with the program variables and so we define breakpoints on program lines or variables. Line breakpoints are treated in a similar way to other debuggers. Here we will concentrate on variable spying and information visualization which is characteristic of TyCO.

Listing 7: Spying Variables with the TyCO Debugger

```
tycodb version 0.1
spying file: program.mil

> run
Line 1:     new x
            ^

> spy x
Line 6:     Print[] |       // this process goes to the run-queue

> n
```

```
Line 6:     x![1] |         // this message is placed at channel x
            ^
variable x changed
> n
Line 6:     x![2] |         // this message is placed at channel x
            ^
variable x changed
> n
Line 6:     x![3] |         // this message is placed at channel x
            ^
variable x changed
> n
Line 4:     x ? (y) = io!printi[y]      // process Print[] is executed
            ^                           // places this object at channel x
variable x changed

> c
program output: 1 2 3
program exited normally
```

Listing 8 shows a small part of the above session where we visualize the contents of the channel x during the execution. Notice that the status is negative if there are messages in the channel, positive if there are objects in the channel or 0 if the channel is empty. The modulus of the value is the number of items in the channel.

Listing 8: Visualization of Channels

```
DebugTyco version 0.1
Spying file: program.mil

...

variable x changed
> n
Line 6:     x![3] |         // this message is placed at channel x
            ^
variable x changed

> print status x

variable x has 3 message(s)
> print queue x

  Frame[3] ->
       Frame[0] = Integer:1
  Frame[2] = null
  Frame[1] = Integer:0
  Frame[0] ->

       Frame[3] ->
            Frame[0] = Integer:2
       Frame[2] = null
       Frame[1] = Integer:0
       Frame[0] ->

            Frame[3] ->
                 Frame[0] = Integer:3
            Frame[2] = null
            Frame[1] = Integer:0
            Frame[0] = null

> n
Line 4:     x ? (y) = io!printi[y]      // process Print[] is executed
            ^                           // places this object at channel x
variable x changed

> print status x

variable x has 2 message(s)

> print queue x

  Frame[3] ->
       Frame[0] = Integer:2
  Frame[2] = null
  Frame[1] = Integer:0
  Frame[0] = ->
```

```
        Frame[3] ->
             Frame[0] = Integer:3
        Frame[2] = null
        Frame[1] = Integer:0
        Frame[0] = null

> print run-queue

  Frame[3] ->
        Frame[0] = Integer:1
  Frame[2] = null
  Frame[1] = Label: val        // the method in the Print process
  Frame[0] = null

> c
program output: 1 2 3
program exited normally
```

Notice how the execution of process Print consumes one message from the channel x. This message, combined with the object that forms the body of Print, reduces and forms another activation record that is placed in the run-queue of the run-time system.

5 Related Work

There are a few implementations of programming languages based on process calculi. These languages have been evolving towards a distributed setting since their base calculi, usually some for of the asynchronous π-calculus, have been extended with the notion of locality, an abstraction that can be mapped, for example, onto virtual machines, IP nodes and network domains.

Rather than go through each one of these programming languages, here we present those most related to TyCO. In particular we focus on the compilation strategy, on the availability of tools for debugging and the level of abstraction that they provide. The programming and debugging tools available are, to our knowledge, representative of the full spectrum of current implementations of process calculi based languages.

Pict [6] is a pure concurrent programming language based on the asynchronous π-calculus. The run-time system is based on Turner's abstract machine [23] specification and the implementation borrows many features from the C programming language. The basic programming abstractions are processes and channels. As usual, processes communicate by sending values on shared channels. Pict deals with recursion by allowing replication only on input processes. Thus, reduction is split into two rules:

$$x?(\tilde{y}).P \mid x![\tilde{v}] \rightarrow \{\tilde{v}/\tilde{y}\}P$$
$$x?*(\tilde{y}).P \mid x![\tilde{v}] \rightarrow x?*(\tilde{y}).P \mid \{\tilde{v}/\tilde{y}\}P$$

Objects in Pict are persistent with each method implemented as an input process held at a distinct channel. The execution of methods in concurrent objects by a client process is achieved by first interacting with a server process (that serves requests to the object and ensures mutual exclusion), followed by the method invocation proper. This protocol [27] for method invocation involves two reductions as opposed to one in TyCO that uses branching structures (records of labelled processes we call objects).

Pict uses replication to model recursion. In Turner's machine the use of replication is controlled by artificially changing the reduction rules so that replication only occurs in response to a message sent to a replicated input process. TyCO's branching structures allow a cleaner definition of reduction since recursion can only occur after communication, in the body of methods, by creating a new instance of the object. Also, replication (as opposed to recursion) generates significant amounts of computational garbage in the form of unused channels and process closures.

The Pict compiler was developed in Objective Caml. The output of the compiler is C code, that, when compiled and linked with extra libraries (the run-time), produces the final executable. Pict does not provide any debugging tool other than the C level tools such as gdb. This exposes the final compiled program, rather than the source, to the programmer and requires from the later some knowledge of the compilation scheme and of the run-time system.

Join [10] and **Jocaml** [5] implement the Join-calculus [9]. Channels (also called ports in Join), expressions and processes are the basic abstractions. Join programs are composed of processes, communicating asynchronously on channels and producing no values and, expressions evaluated synchronously and producing values. Join introduces a powerful extension of the notion of channel: the *join-pattern*. Patterns combine channels, input processes and replication into a single construct. A join-pattern defines a synchronization pattern between input processes waiting on a sequence of channels. For example:

Listing 9: A Join Pattern

```
let subject(s) | verb(v) | object(o) =
    {print_string(s^" "^v^" "^o) ; print_newline() ; }
spawn subject("john doe") | verb("debugged") | object("this program")
```

produces "john doe debugged this program", since the body of the join-pattern is triggered by the three messages sent with spawn. Both Join and Jocaml have some fairly advanced tools for modular software development inherited from its development language, Objective Caml. These include a source to byte-code object compiler, a linker to add compiled modules and a run-time byte-code interpreter. Neither Join nor Jocaml provide any debugging tools specific to the language. Programmers can only use debugging tools available for OCaml.

X-Klaim [15] is based on the KLAIM (Kernel Language for Agents Interaction and Mobility) process calculus [8]. KLAIM uses a communication model akin to the Linda shared tuples [4]. The communication between processes (located at some node l) is performed through the use of asynchronous primitives that manipulate tuples. The basic operations on tuples are: out, in and read.

out (t)@l, evaluates a tuple t and writes a tuple with the value in the tuple-space; in (t)@l, computes a term t and finds a matching tuple t' in the tuple space. The matching operation unifies variables in t and t' and these tuples are removed from the tuple-space; finally, read (t)@l, is similar to in except for the fact that the matching tuple t' is not removed from the tuple space. Thus, the primitives in and read are responsible for the reduction process in X-KLAIM.

For example, if a tuple (!s,"debugged",!o) exists in the tuple-space, the instruction:

```
in(("john doe",!v,"this program"))
```

will result in the variables s, v and o being instantiated with the values "john doe", "debugged" and, "this program", respectively. The tuple (!s,"debugged",!o) is removed from the tuple-space.

The X-Klaim compiler produces Java code that is fed, together with a package KLAVA, to the Java compiler. The KLAVA package provides the run-time system for X-Klaim. The final output is a Java executable that runs on top of the JVM. Currently, there are no tools available to debug X-Klaim programs. Debugging can only be pursued at the level of the final Java executable.

6 Conclusions

In this paper we have described the implementation of a debugger for the TyCO programming language. The language is based on the TyCO process calculus and implements a new paradigm for programming: the mobility paradigm. The main abstractions are processes (computations) and channels (e.g., references, links, sockets). Processes compute by exchanging channels and it is this mobility of channels that provides the computational power to the calculus. Channels may be used to program both datatypes and communication patterns between processes.

In this setting, the TyCO programming language [24] provides a framework on which to experiment with a new programming paradigm. The usefulness of the language as a teaching tool was hampered by the lack of a development environment and, more specifically, of a debugger. With this work we have added this functionality to the language and made it more appealing for programmers.

The debugger we have developed is implemented as a straightforward extension of the MIL virtual machine through its external services and its compiler support was easily integrated into the TyCO compiler. The version of TyCO for which the debugger was developed is single-threaded. Concurrency and non-determinism may be simulated in this version by compiling parallel processes in random sequences or, better still, by transforming the run-queue of activation records into a pool and randomizing the way it is accessed. This will produce distinct run-time flows for the same program.

The TyCO virtual machine was extended during the period of this work to support full concurrent behavior. In its current version, the virtual machine starts a set of concurrent threads, each of which with its private set of registers and run-queue. There is also a global run-queue through which threads may exchange activation records. The threads interact by accessing shared channels in the shared address space. Thus, the next logical step in the development of the debugger will be to support multithreaded execution, enabling the user to view the execution of the program from the perspective of any of the threads involved.

Later, another tool we envision of great importance for the productivity of the debugger is a graphical interface. In this way, the run-time data-structures associated with the program can be visualized at any given time in a more intuitive way. We purposely designed the debugger with GNU's gdb as a reference since this allows a seamless integration with powerful tools such as ddd [7] or simple GUIs like xxgdb [28] or kdbg [14].

References

[1] M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: the Spi-Calculus. In *Computer and Communications Security (CCS'97)*, pages 36–47. The ACM Press, 1997.

[2] E. Berk and C. Ananian. Jlex: lexical analyzer generator, 1996. http://www.cs.princeton.edu/ appel/modern/java/JLex/.

[3] G. Boudol. Asynchrony and the pi-calculus. Technical Report 1702, INRIA, Institut National de Recherche en Informatique et en Automatique, 1992.

[4] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, 1989.

[5] S. Conchon and F. Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *ASA/MA'99*, pages 22–29. IEEE Computer Society, 1999.

[6] D. Turner, B. Pierce. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling and M. Tofte, editor, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[7] DDD - Data Display Debugger. http://www.gnu.org/software/ddd/.

[8] R. DeNicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[9] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385. ACM, 1996.

[10] C. Fournet and L. Maranget. *The Join-Calculus Language (release 1.02)*. Institute National de Recherche en Informatique et Automatique, 1997.

[11] GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/gdb.html.

[12] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of the ECOOP '91 European Conference on Object-oriented Programming*, LNCS 512, pages 133–147. Springer-Verlag, 1991.

[13] S. Hudson, F. Flannery, and C. Ananian. Cup parser generator, 1996. http://www2.cs.tum.edu/projects/cup/.

[14] KGDB: Linux Kernel Source Level Debugger. http://kgdb.linsyssoft.com/.

[15] R. Pugliese L. Bettini, R. De Nicola. X-Klaim and Klava: Programming Mobile Code. *TOSCA 2001, Electronic Notes on Theoretical Computer Science, Elsevier*, 62, 2001.

[16] L. Lopes, V. Vasconcelos, and F. Silva. Fine grained multithreading with process calculi. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 217–226. IEEE Computer Society Press, October 2000.

[17] R. Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.

[18] R. Milner. *Communication and Concurrency*. Preentice-Hall, 1989.

[19] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[20] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[21] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *LICS98*, pages 176–185. IEEE Computer Society Press, 1998.

[22] H. Paulino, P. Marques, L. Lopes, V. Vasconcelos, and F. Silva. A Multi-Threaded Asynchronous Language. In *7th International Conference on Parallel Computing Technologies (PaCT'03)*, volume 2763 of *LNCS*, pages 316–323. Springer-Verlag, 2003.

[23] D. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

[24] Typed Concurrent Objects. http://www.ncc.up.pt/tyco/.

[25] V. Vasconcelos. Typed Concurrent Objects. In *European Conference on Object-Oriented Programming (ECOOP'94)*, volume 821 of *LNCS*, pages 100–117. Springer-Verlag, 1994.

[26] V. Vasconcelos and L. Lopes. A Multi-threaded Typed Assembly Language: Intermediate Language and Virtual Machine. Unpublished.

[27] D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.

[28] Xxgdb. ftp://ftp.x.org/contrib/utilities/.