

Report on *Types as Models: Model Checking Message-Passing Programs*

Austin Parker

November 19, 2002

Report

The paper *Types as Models: Model Checking Message-Passing Programs*¹ can be divided into three main themes: introducing formalism, using formalism to prove modeling theorems, and showing a program implement these modeling theorems to give useful results. The formalism introduced works with CCS processes, an environment in which one may model concurrent systems. The formalism allows manipulation and comparison of the actions available to a particular CCS process at a particular time. Also used in this paper is the pi-calculus, a system much stronger but formally very similar to CCS. The systems differ in that for a pi-calculus process, different threads are allowed to pass information to one another, whereas with CCS processes, each thread is only able to determine that a communication has taken place, there is no transmission of information. CCS processes are therefore simpler, and significantly easier to model. Using the formalism, the authors are able to create a program, called PIPER, which can automatically check if a given CCS process properly abstracts another given pi or CCS process. The idea is that once we have abstracted the CCS processes, we can model check only the smaller CCS processes and be assured that the results will be applicable to the large original pi-processes.

The major theorems given in the paper are the theorems relating to process subtyping, the Assume-Guarantee principle, and the theorems regarding the use of typed channels for pi-process simulation. Process subtyping and the assume-guarantee principle both work through formal manipulation of traces of CCS processes. A trace of a CCS process is a set of actions changing the CCS process, for instance, in:

$$(\nu x, y)(x!^1.y?^2|x?^3.y!^4) \xrightarrow{x^{1,3}} (\nu y)(y?^2|y!^4) \xrightarrow{y^{4,2}} \epsilon$$

$x^{1,3}y^{4,2}$, and $x^{1,3}$ would both be considered traces of $(\nu x, y)(x!^1.y?^2|x?^3.y!^4)$. Each individual move in a trace is called an action. One can have actions $x!^t$,

¹hereafter TM. Authors: Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Can be found on the web at <http://research.microsoft.com/behave/>

y^{t_1, t_2} or $a^{?t}$ (x , y and a are called channel names). The authors consider ω to be a trace of CCS process Γ , with $\omega_{[i]}$ as the i^{th} action in the trace, and define the operation ω° by “replacing all actions of the form x^{t_1, t_2} , $x^{!t}$, $x^{?t}$ with x ” (section 2.2 of TM). I quote them directly here because it seems to me that this definition allows worrisome things to happen later.² Also defined is the operation ω^τ , which removes all τ actions (the concept of the hidden τ action is never explicated in this paper, and appears to have little bearing on the results). Then the authors define a norm on traces as $norm(\omega) = |\omega| = (\omega^\tau)^\circ$. $=_N$ is defined on traces ω and ω' by $|\omega| = |\omega'|$ if and only if $\omega =_N \omega'$.

These operations allow the authors to define *open simulation* as follows for CCS processes Γ, Δ, Θ .

$$\Gamma \leq_\Theta \Delta \iff \forall \sigma. \forall a. \sigma(\Gamma) \xrightarrow{a} \Gamma' \Rightarrow \exists \Delta'. \sigma(\Delta) \xrightarrow{(a)_\Theta} \Delta' \text{ and } \Gamma' \leq_\Theta \Delta'$$

σ is a variable re-naming function and can be considered the identity for all intuitive purposes (I imagine it has an effect on how PIPER is coded). This is the definition of open simulation given in the paper. The intuitive conceptualization of this is to see process Δ as being able to do anything that process Θ does in any given run of process Γ . So if $\Gamma = \Delta | \Upsilon$ then $\Gamma \leq_\Delta \Delta$ is trivially true (because Δ can do anything Δ can do in Γ). $\Gamma \leq \Delta$ is defined as $\Gamma \leq_\Gamma \Delta$. This is the definition given in the paper, but I think it is missing a trivial, but formally important piece. Since it is defined recursively, the base case should somehow be specified. From my understanding of the paper, I would guess that the authors would intend a definition which replaced $\Gamma' \leq_\Theta \Delta'$ with *either* $\Gamma' \leq_\Theta \Delta'$ *or* $\Gamma = \Gamma'$ *and* $\Delta = \Delta'$. This addition to the definition would take care of both the recursive cases and the simple reducible cases.

This idea of open simulation among CCS processes is important in the assume guarantee principle, a central concept of the paper. The idea is that if Γ' simulates part of a CCS process, and Δ' simulates the rest of the processes, then Γ' and Δ' together should be able to simulate the entire CCS process. Formally, this is stated:

For $\Gamma, \Gamma', \Delta, \Delta'$ if

1. $(\eta\vec{x})(\Gamma|\Delta') \leq_\Gamma \Gamma'$
2. $(\eta\vec{x})(\Gamma'|\Delta) \leq_\Delta \Delta'$
3. *non-blocking condition*

then

$$(\eta\vec{x})(\Gamma|\Delta) \leq (\eta\vec{x})(\Gamma'|\Delta')$$

Where the *non-blocking condition* states that along channels \vec{x} , there is a send for every receive and a receive for every send (for a more specific wording of this condition, see the paper). Theoretically, all paths possible in $(\eta\vec{x})(\Gamma|\Delta)$ are available in $(\eta\vec{x})(\Gamma'|\Delta')$, but since we can work things such that the latter

²See section labeled *Problems?*.

process is much smaller than the first, we should be saving ourselves quite a bit of time.

One might wonder how we are, in the end, going to simulate pi-processes with CCS processes. This occurs through *bounded type signatures*. There are two types of bounded signatures available in PIPER's system, effect types and symmetric signatures. Each is predicated on a channel, for instance, $x : ch(\vec{y} : C)\langle\Gamma\rangle$. x is a channel of type $ch(\vec{y} : C)\langle\Gamma\rangle$. The authors use such typing information by forcing one (through the type system) to allow a concurrent version of Γ to run whenever a send occurs on x . This effectively simulates a pi-process' ability to send a channel over which a predetermined action will occur. For instance, say that a client and a server connect over a channel x . The server then sends another channel's name over x and the client proceeds to open the received channel and receive a file. Channel x can be said to have type $(y : receive_file_ch)\langle y?\rangle$ because whenever there is a send on x a CCS process which does $y?$ is opened. The pi-process for the above interaction is:

$$\begin{aligned} Client(x) &= x?[y].y? \\ Server(x) &= (\nu a)(x![a].a!) \\ Main_\pi &= (\nu b)(Client(b)|Server(b)) \end{aligned}$$

That process is simulated by a CCS process using an effect type on x . x is of type $ch(y : receive_file_ch)\langle y?\rangle$ in the following CCS process simulating $Main_\pi$.

$$\begin{aligned} Client(x) &= x? \\ Server(x) &= (\nu a)(x!.(a!|y?\{y \mapsto a\}))^3 \\ Main_{CCS} &= (\nu b)(Client(b)|Server(b)) \end{aligned}$$

You will notice that the CCS version is less intuitive than the pi version (because it seems that the main job of the client is now done in the Server procedure). But you should also notice that the two are equivalent. These are effect channels, and it could be noted that they are limited in their ability to model two-way communication over a passed channel. That is, only the client's actions are built into the channel type, the server is free to do whatever it likes over a channel regardless of type. To solve this problem, the authors developed symmetric channels of the form $ch(\vec{y} : C)\langle\Gamma \Rightarrow \Delta\rangle$. The idea is that the server's actions are specified by Γ and the clients by Δ (or vice versa). Barring some name changes used to contain the actions of Γ and Δ to one another, the CCS simulation of a send over a channel of this type results in the spawning of the process $(\Gamma|\Delta)$. The authors build a type system around these types and give type rules which (I assume) are used to check input into the PIPER program.

The authors also give some examples of systems checked by the PIPER program, and come up with apparently positive results. I was disappointed,

³ $y?\{y \mapsto a\}$ is equivalent to $a?$. The operation $\Gamma\{\vec{x} \mapsto \vec{z}\}$ replaces all \vec{x} in Γ with \vec{z} simultaneously.

however, that there were not numbers showing me that PIPER actually did things faster than an equivalent system not using the assume-guarantee principle and channel typing. Theoretically it seems that PIPER should be better in that sense, but practice with theory leads one to mistrust such results in practice until they are seen. I am also disappointed that PIPER was not available. It must either be in an early beta stage or be proprietary or a real pain to document.

This paper is interesting because it uses real, non-trivial (and therefore interesting) formal manipulation to produce results which are useful. There were some places in the paper where I wonder if the authors are going to fine tune their formalism, but it seems to do the job. There are two things which I feel would add to this paper: versions of the proof which do not rely on the reader's intuition (even the versions available in the technical report leave much of the proof as a proverbial 'exercise for the reader'), and hard physical data from actual runs of PIPER. Overall though, the paper is interesting and is engaged in a deep subject to which one could legitimately and happily devote time.