WORST-CASE INPUT GENERATION FOR CONCURRENT PROGRAMS UNDER NON-MONOTONE RESOURCE METRICS

LONG PHAM AND JAN HOFFMANN

Carnegie Mellon University
e-mail address: longp@andrew.cmu.edu, janh@andrew.cmu.edu

Abstract. Worst-case input generation aims to automatically generate inputs that exhibit the worst-case performance of programs. It has several applications, and can, for example, detect vulnerabilities to denial-of-service attacks. However, it is non-trivial to generate worst-case inputs for concurrent programs, particularly for resources like memory where the peak cost depends on how processes are scheduled.

This article presents the first sound worst-case input generation algorithm for concurrent programs under non-monotone resource metrics like memory. The key insight is to leverage resource-annotated session types and symbolic execution. Session types describe communication protocols on channels in process calculi. Equipped with resource annotations, resource-annotated session types not only encode cost bounds but also indicate how many resources can be reused and transferred between processes. This information is critical for identifying a worst-case execution path during symbolic execution. The algorithm is sound: if it returns any input, it is guaranteed to be a valid worst-case input. The algorithm is also relatively complete: as long as resource-annotated session types are sufficiently expressive and the background theory for SMT solving is decidable, a worst-case input is guaranteed to be returned. A simple case study of a web server's memory usage demonstrates the utility of the worst-case input generation algorithm.

1. Introduction

Understanding the worst-case performance of programs and when it is triggered helps programmers spot performance bugs and take preemptive measures against algorithmic complexity attacks. One line of research on worst-case input generation employs symbolic execution (e.g. WISE [BJS09] and SPF-WCA [LKP17]). These techniques first symbolically execute a program on a small input to identify a worst-case execution path. This path is then generalized and used to guide the symbolic execution of a larger input. These techniques do not explore the entire search space of large inputs. So they are scalable but unsound.

Moreover, no prior works investigate worst-case input generation in the joint setting of (i) concurrent programming and (ii) non-monotone resource metrics (e.g. memory). A resource metric is *non-monotone* if resources can be freed up as well as consumed. Worst-case input generation for this setting has a practical value. For example, denial-of-service (DoS) attacks overwhelm the memory of servers, which are typically concurrent programs. Hence, the

Key words and phrases: Worst-case input generation, resource analysis, session types, concurrent programming, amortized analysis.

Preprint submitted to
Logical Methods in Computer Science

© L. Pham and J. Hoffmann
Creative Commons

worst-case input generation for concurrent programs under non-monotone resource metrics can identify vulnerabilities to DoS attacks.

This article presents the first sound worst-case input generation algorithm for message-passing concurrent programming under non-monotone resource metrics. Our work builds on the type-guided worst-case input generation for functional programming by Wang and Hoffmann [WH19]. In their algorithm, symbolic execution is guided by *resource-annotated types* inferred by automatic amortized resource analysis (AARA) [HJ03, HH10, HAH12]. AARA is a fully automatic type-based resource analysis technique for functional programming. Resource-annotated types encode the amount of available resources. To identify a worst-case execution path, the algorithm searches for an execution path where the cost bound is tight. Thanks to the soundness of AARA, whenever Wang and Hoffmann's algorithm generates a candidate worst-case input, it is guaranteed to be a valid worst-case input.

To extend Wang and Hoffmann's work to message-passing concurrent programming, we must first adapt the notion of *skeletons*. They specify the shapes and sizes of worst-case inputs to be generated. In designing skeletons for message-passing concurrent programming, we face the following challenges nonexistent in functional programming:

- Interaction between channels: the shapes of inputs on different channels may be dependent on one another. This stands in contrast to functional programming where inputs' shapes are independent of one another.
- Co-inductive interpretation: inputs to a concurrent program may be infinite.
- Intertwining of input and output: input and output are intertwined, unlike in functional programming where an entire input is provided before execution.

Our first contribution is to design a suitable notion of *session skeletons* for message-passing concurrent programming. Session skeletons are built on *session types* [Hon93] that describe communication protocols on channels in process calculi.

In session-typed concurrent programming, monotone costs like work (i.e. sequential running time) are independent of how concurrent processes are scheduled. Hence, monotone costs in message-passing concurrent programming can be treated in the same manner as in Wang and Hoffmann's work for functional programming. Meanwhile, costs like span (i.e. parallel running time) are dependent on schedules, but they are outside the scope of this article. Instead, we are interested in work-like, non-monotone costs such as memory.

Under non-monotone resource metrics, there are two types of costs: *net cost* and *high-water-mark cost*. The net cost is the net quantity of resources consumed, and the high-water-mark cost is the peak net cost that has been reached. In session-typed concurrent programming, while the net cost of a program is independent of processes' schedules, the high-water-mark cost is dependent on the schedules. Moreover, the operational semantics of concurrent programming languages do not specify the exact scheduling of processes. As a consequence, haphazardly executing a concurrent program does not necessarily reveal the correct high-water-mark cost. We additionally need to know the worst-case schedule of processes. However, it is non-trivial to learn such a schedule on the fly during program execution, unless we preprocess the program beforehand. Thus, Wang and Hoffmann's algorithm cannot directly be extended to message-passing concurrent programming under non-monotone resource metrics.

To handle high-water-mark costs' dependency on schedules, we leverage *resource-annotated session types*. AARA has been integrated into session types, yielding resource-annotated session types [DHP18]. These types capture both (i) cost bounds on high-water-mark costs and (ii) how many resources can be reused and transferred between processes. Thanks to the availability of cost bounds, like Wang and Hoffmann's algorithm, our worst-case input generation algorithm is sound. Also, the information about resource transfer enables us to correctly track high-water-mark costs during symbolic execution.

Our contributions are listed below.

- We present the first sound worst-case input generation algorithm for message-passing concurrent programming under non-monotone resource metrics (e.g. memory).
- We propose a suitable notion of skeletons. We also address several technical challenges posed by session types in the design of skeletons.
- We have proved the soundness (i.e. if the algorithm returns anything, it is a valid worst-case input) and relative completeness (i.e. if AARA is sufficiently expressive and the background theory for SMT solving is decidable, a worst-case input is guaranteed to be returned).
- We present a case study of worst-case input generation for a web server's memory usage.

The article is structured as follows. Section 2 provides an overview, describing (i) the challenge of generating worst-case inputs for concurrent programs under non-monotone resource metrics and (ii) how we overcome this challenge. Section 3 presents resource-aware SILL, a message-passing concurrent programming language equipped with resource-annotated session types. Section 4 defines skeletons and describes the challenges in their design. Section 5 presents a worst-case input generation algorithm guided by resource-annotated session types. Section 6 demonstrates the algorithm through a case study of a web server. Finally, Section 7 discusses related work, and Section 8 concludes the article.

2. Overview

Processes and Channels. This work uses the message-passing concurrent programming language SILL [CP10, TCP13, PG15]. Suppose we are given a process P with two channels c_1 and c_2. Communication on c_1 and c_2 can be bidirectional. Process P uses c_1 as a client and provides c_2. This process is depicted in Figure 1 (a). The environment that P interacts with is called *the external world*. Figure 1 (b) depicts a general SILL program consisting of multiple concurrent processes. The network of concurrent processes in SILL must be an acyclic graph (i.e. a tree)[1].

We use the following scenario as a running example. On c_1, IP addresses of packets are sequentially sent from the external world to P. The stream of IP addresses may be finite or infinite. Given the input stream on c_1, process P counts occurrences of each IP address. Once c_1's input stream terminates, P outputs on channel c_2 the number of IP addresses with at least four occurrences.

Suppose P is implemented as follows. P maintains a key-value store where keys are IP addresses and values are the numbers of occurrences. When a new IP address is encountered, it is added as a key to the key-value store, incurring the memory cost of 2. It is because

[1]SILL's type system was designed to guarantee deadlock freedom. Because a cycle of channels may cause a deadlock, processes' network must be acyclic. Although some cycles of channels are benign, SILL's type system is not sophisticated enough to handle them. [BP17, BTP19] present more fine-grained type systems for deadlock freedom.

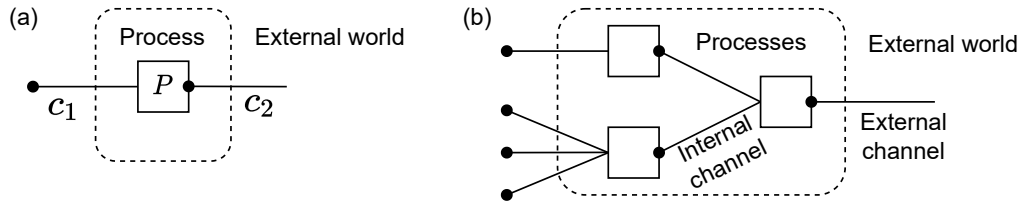

Figure 1: (a) Process P uses channel c_1 as a client and provides channel c_2. A dot on a channel denotes the channel's provider. (b) General SILL program consisting of multiple concurrent processes. An internal channel connects two processes; an external channel connects a process with the external world.

we need one memory cell for the key and another for the value. When c_1's input stream terminates, all memory is released.

Session Types. Session types [Hon93] describe communication protocols on channels. The communication on c_1 is described by the session type

$$\mu X. \oplus \{\mathsf{cons} : \triangleright^2 \mathsf{int} \wedge X, \mathsf{nil} : \mathbf{1}\}. \tag{2.1}$$

μX denotes a recursive session type with a type variable X. The type constructor $\oplus$ means internal choice, that is, c_1's provider (i.e. the external world) chooses between labels cons and nil and sends the choice. If cons is chosen, a value of type int is sent by the external world, and we recurse back to X. Conversely, if nil is chosen, c_1 is closed. The resource annotation $\triangleright^2$ will be explained shortly.

The session type of c_2 is $\mathsf{int} \wedge \mathbf{1}$. It means the provider of channel c_2 sends an integer and then closes the channel by sending the end message.

Resource Annotations. Resource-aware SILL [DHP18] incorporates resource annotations into session types. These resource annotations indicate the amount of *potential*[2] necessary to pay for computational cost. In our example, when a previously unseen IP address is encountered, two memory cells are allocated. Therefore, in the worst case, we need two units of potential to process each IP address on c_1. This is why the resource-annotated type of c_1 (2.1) contains $\triangleright^2$. It denotes that two units of potential are transferred from the channel client (i.e. the external world) to the channel provider (i.e. P).

Resource-annotated types are inferred automatically. Because all constraints generated during type inference are linear, they can be solved by an off-the-shelf linear-program (LP) solver [DBHP19]. Furthermore, the type inference is sound: inferred cost bounds are guaranteed to be valid upper bounds on high-water-mark costs.

[2]Potential and resources sound similar and are sometimes interchangeable. Strictly speaking, potential is an *abstract* resource used in the potential method of amortized analysis [Tar85], on which resource-aware SILL's type system is based. Resources, on the other hand, refer to *concrete* computational resources such as time and memory.

Session Skeletons. A SILL program is a network of processes that interact with the external world. Therefore, an input to a SILL program is a collection of incoming messages from the external world. The incoming messages may be intertwined with outgoing messages produced by the program.

We use the high-water-mark cost, instead of net costs, to define worst-case inputs. This definition of worst-case inputs for memory subsumes the definition of worst-case inputs for running time. For running time, the high-water-mark cost is always equal to the net cost.

The first step in worst-case input generation is to provide a *skeleton* for each external channel. A skeleton is a symbolic input containing variables, and their concrete values are determined later. Skeletons specify the shape of worst-case inputs to be generated.

For channel c_1 in the example, a possible skeleton is

$$\oplus\{\mathsf{cons} : \triangleright^2 x_1 \wedge \cdots \oplus \{\mathsf{cons} : \triangleright^2 x_{10} \wedge \oplus\{\mathsf{end} : \mathbf{1}\}\} \cdots\}, \tag{2.2}$$

where $x_1, \ldots, x_{10}$ are variables. This skeleton specifies that the external world should send ten cons's followed by nil. As channel c_2 does not take in any input from the external world, c_2's skeleton need not specify the shape of an input.

Skeletons must satisfy the following requirements:

- A skeleton must be compatible with its associated session type. This compatibility relation coincides with the subtyping relation [GH05].
- The input portion of a skeleton must be finite.

Because P allocates two memory cells whenever a new IP address is encountered, a worst-case input contains mutually distinct IP addresses on c_1's input stream. An example worst-case input to P that conforms to the skeleton (2.2) is

$$\forall 1 \leq i \leq 10. x_i = i. \tag{2.3}$$

Symbolic Execution. Thanks to the soundness of resource-aware SILL, worst-case input generation boils down to finding an input whose high-water-mark cost achieves the cost bound. To this end, we symbolically execute a program on a skeleton. During symbolic execution, we search for an execution path where the program's potential reaches zero since the last time potential was supplied by the external world.

This strategy correctly identify a worst-case input. Suppose for simplicity that all necessary potential is supplied at the start of execution (Figure 2 (a)). Then the cost bound for memory is tight if and only if the potential reaches zero at some point. If the potential never reaches zero, we can lower the cost bound while covering all computational cost (and without plunging into negative potential). Hence, the cost bound is not tight.

In SILL, potential is supplied to a program gradually, rather than all at once. Hence, whenever the external world supplies potential to the program, we must ensure that the program will eventually reach potential zero (Figure 2 (b)). Otherwise, we could lower the cost bound (Figure 2 (c)) without plunging into negative potential.

High-Water Marks under Concurrency. In the presence of multiple processes, their concurrency poses a challenge: different schedules for symbolic execution may result in different high-water marks of non-monotone resources. Generally, in concurrent programming, monotone resources also have dependency on schedules. An example is a race condition where two processes compete for a single message and their monotone cost depends on which process wins. However, in session-typed concurrent programming like SILL, session

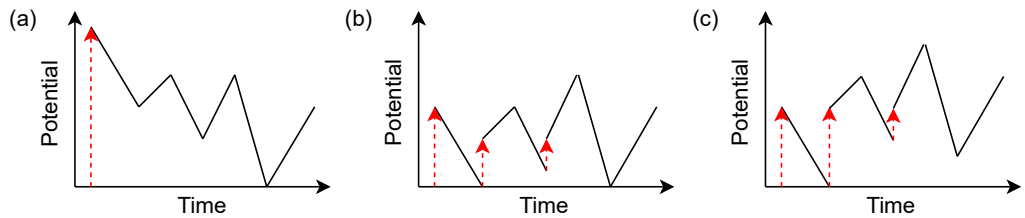

Figure 2: (a) Tight cost bound when all potential is supplied at once. The red dashed arrow indicates the potential supplied by the external world. (b) Tight cost bound when potential is supplied gradually. (c) Loose cost bound. Potential zero is never reached after the last injection of potential. So we can lower the cost bound (i.e. shorten the second and third red dashed arrows) without plunging into negative potential.

types make communication more rigid. As a result, the above example never arises in SILL, making monotone resources independent of schedules.

To illustrate the dependency of non-monotone resources on schedules, consider two processes, P_1 and P_2. Initially, these two processes run independently. P_1 has the high-water mark $h = 4$ and net cost $w = 0$. Also, P_2 has $(h, w) = (1, 0)$. Next, P_1 sends a message to P_2, thereby synchronizing them. We assume that the communication between P_1 and P_2 is *asynchronous*. That is, once it sends a message, P_1 does not need to wait for P_2 to receive the message. After sending the message, P_1 incurs $(h, w) = (2, 0)$. After receiving the message, P_2 incurs $(h, w) = (3, 0)$. This situation is depicted in Figure 3 (a). In the figure, tick q, where $q \in \mathbb{Q}$ is a rational number, means q units of resources are consumed. As a special case, if $q < 0$, then tick q means $|q|$ units of resources are freed up. The arrows indicate the happened-before relation [Lam78].

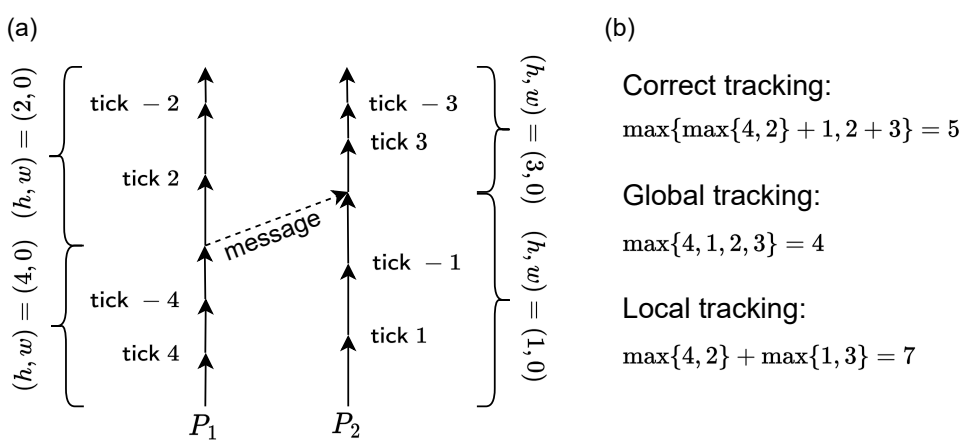

Figure 3: (a) Concurrent processes P_1 and P_2. Time passes by in the direction of arrows. (b) Results of global tracking and local tracking.

What is the combined high-water-mark cost of P_1 and P_2? Before P_2 receives a message, their combined high-water mark is $\max\{4, 2\} + 1 = 5$. Due to the asynchrony of

communication, by the time P_2 receives the message, P_1 may have just finished the first phase where $(h, w) = (4, 0)$ or may already be in the second phase where $(h, w) = (2, 0)$. Therefore, the high-water mark of P_1 before P_2 receives the message is given by $\max\{4, 2\}$. The high-water mark of P_2 before it receives the message is $h = 1$. Finally, if the peak net costs of P_1 and P_2 happen at the same time, their combined high-water mark is $\max\{4, 2\} + 1 = 5$.

After P_2 receives a message, the combined high-water mark is $2 + 3 = 5$. So the overall high-water mark throughout the execution is $\max\{5, 5\} = 5$.

To identify a worst-case execution path, we must be able to calculate the correct high-water mark of any execution path. Can we do so on the fly while executing the program? There are two ways to track costs: global and local tracking. In global tracking, we have a global cost counter shared by all processes. In local tracking, each process tracks its own cost. The combined cost is the sum of all local costs after the program terminates.

Neither global tracking nor local tracking returns the correct high-water mark (Figure 3 (b)). On the one hand, global tracking may underestimate it. Before synchronization, if the peak costs of P_1 and P_2 happen at different times, the global counter registers a high-water mark below 5. On the other hand, local tracking may overestimate the high-water mark. When the program terminates, P_1's local counter registers $\max\{4, 2\} = 4$, and P_2's counter registers $\max\{1, 3\} = 3$. Their sum is 7, which is an overestimate.

An innovation for resolving this issue is to transfer *potential* between processes. Suppose P_1 is initially given four units of potential and that P_2 is initially given one unit. In P_1, after it runs tick 4, all potential stored in P_1 is consumed. But tick -4 in P_1 frees up four units of potential. Likewise, P_2's potential is used up by tick 1 but is then freed up by tick -1. When P_1 sends a message, the message also carries two units of potential. This potential is paid by P_1, and in turn, it is used to pay for P_2's cost. Figure 4 (a) depicts this situation.

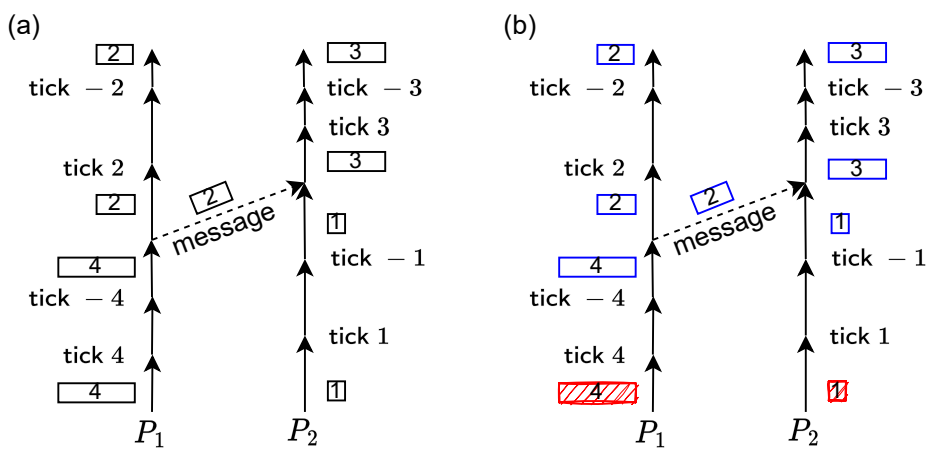

Figure 4: (a) Potential helps us derive the correct high-water mark. A rectangle next to an arrow represents the potential during the arrow's time period. The number inside a rectangle and its length indicate the amount of potential. If no rectangle exists, it indicates zero potential. (b) Same diagram as (a), but potential is colored red and blue.

The combined high-water mark of P_1 and P_2 is bounded above by the total potential supplied at the beginning, namely $4 + 1 = 5$. This bound is tight.

What makes the potential method more powerful than local tracking is the ability to transfer potential. By transferring potential between processes, we are balancing out their local cost counters such that their sum yields the correct combined high-water mark. Previously, in Figure 3 (a), P_2's local cost counter registers $(h, w) = (3, 0)$ after P_2 receives a message from P_1. This is the root cause of overestimation. To fix it, we adjust the counter's reading to $(h, w) = (1, 0)$ by having P_1 send two units of potential to P_2.

Information about potential transfer is exactly what resource-annotated session types offer, in addition to the overall cost bounds. Therefore, by having them guide symbolic execution, we can correctly count high-water-mark costs and identify a worst-case execution path of a SILL program.

Checking Tightness of Cost Bounds. Resource-annotated session types already capture valid bounds on high-water marks. For cost bounds to translate into true high-water marks, it remains to ascertain that the cost bounds are tight. This is done by tracking individual units of potential during symbolic execution.

Let us call (i) the potential supplied by the external world *red potential* and (ii) the potential freed up in a process *blue potential*. If a process executes tick q for $q > 0$ and stores no potential, the external world must supply at least q units of (red) potential to the process. Conversely, if a process executes tick q for $q < 0$, then $|q|$ units of (blue) potential are freed up and become available to the process. A cost bound is the total red potential supplied by the external world.

A cost bound is tight if it satisfies two conditions. Firstly, red potential must be consumed completely. Otherwise, we could lower the cost bound while paying for all computational costs. Using the same P_1 and P_2 from Figure 3, Figure 5 (a) illustrates a situation where red potential is not consumed entirely. P_1 and P_2 are initially given a total of $4 + 2 = 6$ units of red potential. But 0.5 units of red potential are left unconsumed in P_2, suggesting that the cost bound of 6 is not tight.

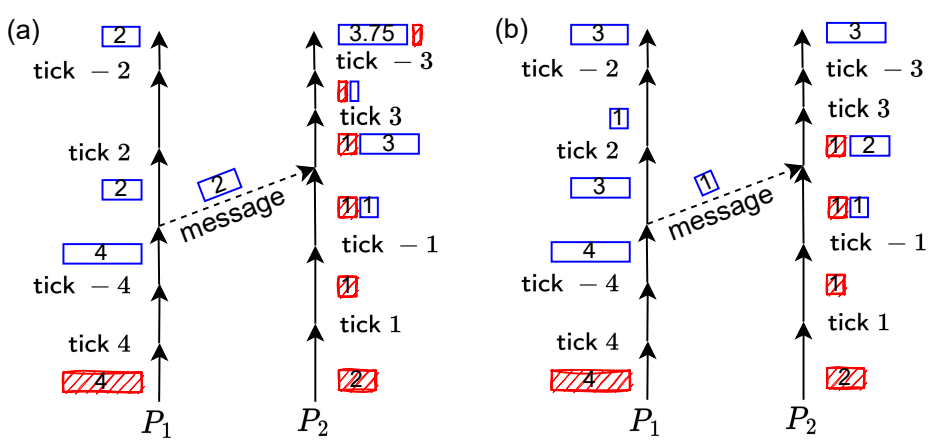

Figure 5: (a) Red potential in P_2 is not entirely consumed. Red hatched rectangles and blue blank rectangles represent red and blue potential, respectively. (b) Blue potential is generated in P_1 before red potential is consumed in P_2. But some blue potential is left unconsumed in P_1.

Secondly, every unit of blue potential must be consumed if its generation precedes the consumption of red potential. Assume otherwise: blue potential is not consumed entirely, while red potential is consumed after that blue potential was generated. An example is illustrated in Figure 5 (b). As in part (a), P_1 and P_2 are initially given four units and two units of red potential, respectively. However, this time, P_1 only sends one unit, instead of two units, of (blue) potential to P_2. Consequently, one unit of the blue potential generated by $\mathsf{tick}\ -4$ on P_1 remains unconsumed in the rest of P_1's lifetime. Furthermore, $\mathsf{tick}\ -4$ happens before red potential is consumed by $\mathsf{tick}\ 3$ on P_2. Hence, we could send more blue potential from P_1 to P_2, thereby substituting the blue potential for red potential supplied to P_2 by the external world. This means the cost bound of 6 is, again, not tight.

Finally, Figure 4 (b) illustrates the case where the above two conditions are met. The cost bound is indeed tight.

Solving Path Constraints. Once a worst-case execution path is identified, its path constraint is fed to an SMT solver to generate a concrete worst-case input. In our example (2.1), the worst-case execution path is where each incoming IP address requires two units of potential. Solving the path constraint of this path, we obtain a set of mutually distinct IP addresses such as (2.3).

3. Resource-Aware SILL

Resource-aware Simple Intuitionistic Linear Logic (SILL) [DHP18] has two constructs: processes and channels. Processes send and receive messages, including channels, on channels. A channel in SILL connects two processes: provider and client. They can communicate in both directions[3]. While each process must provide exactly one channel, the process can be a client of multiple (possibly zero) channels.

Channels are typed with session types. Well-typedness of channels guarantees deadlock freedom (i.e. progress) and session fidelity (i.e. preservation) [CP10].

3.1. Session Types.

Resource-aware SILL has two layers: functional layer and process layer. Types in the functional layer (denoted by τ) and in the process layer (denoted by A) are formed by the following grammar.

$$
\begin{aligned}
b &::= \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid b_1 \times b_2 \mid b_1 + b_2 \\
\tau &::= \{c : A \leftarrow \overline{c_i : A_i} \mathbin{@} q\} && \text{process type} \\
&\quad \mid b \mid \tau_1 \to \tau_2 && \text{functional types} \\
A &::= X \mid \cdots && \text{session types (Table 1).}
\end{aligned}
$$

If a functional term e has a process type $\{c : A \leftarrow \overline{c_i : A_i} \mathbin{@} q\}$, e represents a process that (i) provides a channel c of resource-annotated session type A and (ii) uses channels $c_1, \ldots, c_n$ of resource-annotated session types $A_1, \ldots, A_n$. Throughout this article, a horizontal line above text denotes a vector. The annotation $q \in \mathbb{Q}_{\geq 0}$ indicates the constant potential initially stored in the process.

[3]Even though a provider and a client can communicate in both directions, we still assign different roles to the two endpoints of a channel due to the correspondence between SILL and intuitionistic linear logic [CP10].

Table 1: Resource-annotated session types and process terms. They are described from the viewpoint of channel providers. For the session type $c : \mathbf{1}$, the first row is for c's provider, and the second row is for c's client.

Session type	Cont.	Process term	Cont.	Description
$c : b \supset A$	$c : A$	$x \leftarrow \mathsf{recv}\ c; P_x$	P_x	receive a value of base type b on c and bind it to variable x
$c : b \wedge A$	$c : A$	$\mathsf{send}\ c\ v; P$	P	send a value $v : b$ on channel c
$c : A_1 \multimap A_2$	$c : A_2$	$x \leftarrow \mathsf{recv}\ c; P_x$	P_x	receive a channel of type A_1 on c and bind it to variable x
$c : A_1 \otimes A_2$	$c : A_2$	$\mathsf{send}\ c\ d; P$	P	send a channel $d : A_1$ on c
$c : \&\{\overline{\ell_i : A_i}\}$	$c : A_j$	$\mathsf{case}\ c\ \{\overline{\ell_i \hookrightarrow P_i}\}$	P_j	receive a label on c and do pattern matching
$c : \oplus\{\overline{\ell_i : A_i}\}$	$c : A_j$	$c.\ell_j; P$	P	send a label ℓ_j on channel c
$c : \mathbf{1}$	N/A	$\mathsf{close}\ c$	N/A	close channel c by sending the end message
		$\mathsf{wait}\ c; P$	P	wait for the end message of c's closure
$c : \triangleleft^q A$	$c : A$	$\mathsf{get}\ c\ \{q\}; P$	P	receive $q \in \mathbb{Q}_{>0}$ units of potential on channel c
$c : \triangleright^q A$	$c : A$	$\mathsf{pay}\ c\ \{q\}; P$	P	send $q \in \mathbb{Q}_{>0}$ units of potential on channel c

Given a channel c and a session type A, a judgment $c : A$ means the communication between c's provider and client proceeds according to A. Type A describes the communication protocol from the viewpoint of c's provider.

Session types are summarized in Table 1. We assume a global signature Σ containing definitions of type variables. Type definitions have the form $X = A_X$, where X is a type variable and A_X is a session type that may mention X (hence recursive). Recursive session types are interpreted co-inductively. So communication can last forever. Also, we require recursive session types to be contractive [GH05]. Lastly, recursive session types are regarded equi-recursive. Hence, throughout this article, type variables can be silently replaced by their definitions.

The type inference algorithm automatically determines resource annotations, namely $q \in \mathbb{Q}_{>0}$ in $\triangleright^q$ and $\triangleleft^q$ [DBHP19]. So users do not need to manually provide q.

3.2. **Syntax.** Fix a set $\mathcal{F}(\ni f)$ of function identifiers. A program in resource-aware SILL is a pair (P, Σ), where P is the main process to run. Σ, called a signature, contains type definitions and function definitions. The syntax of functional terms (denoted by e) and processes (denoted by P) is given below.

$$
\begin{aligned}
e &::= x \mid \langle\,\rangle \mid \mathsf{true} \mid \mathsf{false} \mid n \in \mathbb{Z} \mid f \\
&\quad \mid \cdots \mid c \leftarrow P_{c,\overline{c_i}} \leftarrow \overline{c_i} && \text{process constructor} \\
P &::= c \leftarrow e \leftarrow \overline{c_i}; P_c && \text{spawn a process} \\
&\quad \mid c_1 \leftarrow c_2 && \text{forward messages} \\
&\quad \mid \mathsf{tick}\ q; P && \text{consume resources} \\
&\quad \mid \cdots && \text{process terms (Table 1).}
\end{aligned}
$$

Functional term $c \leftarrow P_{c,\overline{c_i}} \leftarrow \overline{c_i}$ encapsulates process $P_{c,\overline{c_i}}$ that provides channel c and uses channels $\overline{c_i}$.

The process $c \leftarrow e \leftarrow \overline{c_i}; P_c$ first spawns a new process denoted by e. The child process provides channel c and uses channels $\overline{c_i}$ as a client. After spawning e, the parent process proceeds with P_c and uses c as a client.

The process $c_1 \leftarrow c_2$ forwards messages between c_1 and c_2 in both directions.

The process $\mathsf{tick}\ q; P$ consumes $q \in \mathbb{Q}$ units of resources. $\mathsf{tick}\ q$ is inserted either manually by a user or automatically according to some resource metric that specifies the cost of each syntactic form. For instance, if a user is interested in memory, allocation of a 64-bit integer is modeled as $\mathsf{tick}\ 64$.

The rest of the syntax is given in Table 1. The value of q in $\mathsf{get}\ c\ \{q\}; P$ and $\mathsf{pay}\ c\ \{q\}; P$ is automatically inferred.

A resource metric is said to be monotone if every $\mathsf{tick}\ q$ satisfies $q \geq 0$. Conversely, if $q < 0$ is allowed, the resource metric is said to be non-monotone. Examples of non-monotone resource metrics are memory (e.g. heap space) and money (e.g. cryptocurrencies transferred by smart contracts). Non-monotone resource metrics subsume monotone ones.

3.3. **Cost Semantics.** The cost semantics of SILL is defined using substructural operational semantics [PS09], which is essentially a multiset rewriting system [CS06]. A program state is represented by a multiset (called a configuration) of predicates. Rewriting rules specify how one configuration transitions to another. Suppose we are given a rewriting rule

$$\frac{I_1 \quad I_2 \quad \cdots \quad I_n}{J_1 \quad J_2 \quad \cdots \quad J_m}$$

where $n, m \geq 0$. If *all* of $I_1, \ldots, I_n$ exist in a configuration, the next configuration is obtained by replacing $I_1, \ldots, I_n$ with $J_1, \ldots, J_m$. Rewriting rules can be applied in any order.

The cost semantics of SILL uses two predicates: $\mathsf{proc}(c, w, P)$ and $\mathsf{msg}(c, w, M)$. Predicate $\mathsf{proc}(c, w, P)$ represents a process P providing channel c. Predicate $\mathsf{msg}(c, w, M)$ means a message M is being transferred across a channel, but has not been received yet. In $\mathsf{proc}(c, w, P)$, $w \in \mathbb{Q}$ tracks the current net cost of the process. The field $w \in \mathbb{Q}$ in $\mathsf{msg}(c, w, M)$ is used when a process terminates and transfers the net cost at that point to another process.

Key rules of the cost semantics are displayed in Figure 6. The grammar of M and the remaining rules of the cost semantics can be found in Appendix A.1.

$$\frac{\mathsf{proc}(c, w, \mathsf{tick}\ q; P)}{\mathsf{proc}(c, w + q, P)}\mathsf{tick} \qquad \frac{\mathsf{proc}(d, w, \mathsf{send}\ c\ e; P) \qquad e \Downarrow v \qquad c' \text{ is fresh}}{\mathsf{proc}(d, w, P[c'/c]) \qquad \mathsf{msg}(c', 0, \mathsf{send}\ c\ v; c' \leftarrow c)}\supset S$$

$$\frac{\mathsf{msg}(c', w_1, \mathsf{send}\ c\ v; c' \leftarrow c) \qquad \mathsf{proc}(c, w_2, x \leftarrow \mathsf{recv}\ c; P_x)}{\mathsf{proc}(c, w_1 + w_2, P_v[c'/c])}\supset R$$

Figure 6: Key rules of SILL's cost semantics. The judgment $e \Downarrow v$ means functional term e evaluates to value v.

In the rule tick, whenever $\mathsf{tick}\ q$ is executed, h and w in the predicate $\mathsf{cost}(h, w)$ are updated. The rule $\supset S$ is for sending a message e on channel c, and the rule $\supset R$ is for receiving a message on channel c. Since rewriting rules can be applied in any order, $\supset R$ is

not necessarily applied *immediately* after $\supset S$. Hence, communication between processes is asynchronous.

The net cost of a configuration C is the sum of w's in $\mathsf{proc}(c, w, P)$ and $\mathsf{msg}(c, w, M)$ in C. The high-water mark of C is its peak net cost that has been reached so far.

3.4. **Type System.** Fix a signature Σ containing type definitions and function definitions. A typing judgment of the process layer is $\Phi; \Delta; q \vdash P :: (c : A)$. Here, Φ is a functional-layer typing context, and Δ is a process-layer typing context that maps channels to resource-annotated session types. The channels in Δ's domain are used by P as a client. $q \in \mathbb{Q}_{\geq 0}$ denotes how much potential is stored in P. Channel c is provided by P and has a resource-annotated session type A.

Figure 7 displays key rules of resource-aware SILL's type system. Remaining rules are in Figure 13 in Appendix A.1.

$$\boxed{\Phi; \Delta; q \vdash P :: (c : A)}$$

$$\frac{\Phi; \Delta; q \vdash P :: (c : A) \qquad p > q}{\Phi; \Delta; p \vdash P :: (c : A)}\,\mathsf{relax} \qquad \frac{\Phi; \Delta; p \vdash P :: (d : D)}{\Phi; \Delta; p + q \vdash \mathsf{tick}\ q; P :: (d : D)}\,\mathsf{tick}$$

$$\frac{\Phi; \Delta, c : A; p \vdash P :: (d : D) \qquad \Phi \vdash e : b}{\Phi; \Delta, c : b \supset A; p \vdash \mathsf{send}\ c\ e; P :: (d : D)}\,{\supset}L \qquad \frac{\Phi, x : b; \Delta, p \vdash P_x :: (c : A)}{\Phi; \Delta; p \vdash (x \leftarrow \mathsf{recv}\ c; P_x) :: (c : b \supset A)}\,{\supset}R$$

Figure 7: Key rules of the type system of resource-aware SILL. $\Phi \vdash e : \tau$ in the rules spawn and $\supset L$ is a typing judgment for the functional layer.

Theorem 3.1 states the soundness of resource-aware SILL's type system. Prior works [DHP18, DBHP19] only prove the soundness under monotone resource metrics. Nonetheless, to extend the soundness result to non-monotone resource metrics, it suffices to re-examine the inductive case for tick.

Theorem 3.1 (Soundness of resource-aware SILL [DHP18, DBHP19])**.** *Given a configuration C, let p be the total potential stored in C. If $C \to^* C'$ (i.e. C' is reachable from C) and h is the high-water-mark cost of C', then $h \leq p$ holds.*

4. Session Skeletons

Skeletons are symbolic inputs specifying the shape of worst-case inputs to be generated. In functional programming, given a function that takes in lists, if we want a worst-case input of length three [WH19], an appropriate skeleton is $[x_1, x_2, x_3]$, where x_i's are variables whose values are determined later.

Message-passing concurrent programming poses three challenges in the design of skeletons. Firstly, in the presence of multiple channels, their skeletons are interdependent due to the interaction between channels. Secondly, input to concurrent programs may be infinite. Thirdly, input and output are intertwined in such a way that output influences the acceptable set of subsequent inputs. Our design of skeletons works around the first two challenges. The last challenge, described in Section 4.4, is beyond the scope of our design.

4.1. **Syntax.** Fix a set $\mathcal{X}_{\text{skeleton}}(\ni x)$ of skeleton variables. They are placeholders for concrete values in a worst-case input and will be determined later. Skeletons are formed by this grammar:

$$
\begin{aligned}
H ::= {} & x \mid \langle\,\rangle \mid \mathsf{bool} \mid \mathsf{false} \mid n \in \mathbb{Z} & \\
& \mid \langle H_1, H_2\rangle \mid \ell \cdot H \mid r \cdot H && \text{skeleton constructors} \\
K ::= {} & b \supset K \mid H \supset K \mid b \wedge K \mid H \wedge K && \text{value input/output} \\
& \mid K_1 \multimap K_2 \mid K_1 \otimes K_2 && \text{channel input/output} \\
& \mid \&\{\overline{\ell_i : K_i}\} \mid \&_x\{\overline{\ell_i : K_i}\} && \text{external choice} \\
& \mid \oplus\{\overline{\ell_i : K_i}\} \mid \oplus_x\{\overline{\ell_i : K_i}\} && \text{internal choice} \\
& \mid X \mid \mathbf{1} \mid \triangleleft^q K \mid \triangleright^q K. &&
\end{aligned}
$$

H is a skeleton for the functional layer, and K is a skeleton in the process layer. K's grammar is similar to that of resource-annotated session types (Section 3.1). A difference is that, in addition to $b \supset K$, we have $H \supset K$, where b is a base type and H is a functional-layer skeleton. $H \supset K$ is used when the input is generated by the external world, whereas $b \supset K$ is used when the input is sent by a process. Likewise, we have two additional constructs, $\&_x\{\overline{\ell_i : K_i}\}$ and $\oplus_x\{\overline{\ell_i : K_i}\}$, with a skeleton variable $x \in \mathcal{X}_{\text{skeleton}}$ in subscripts. These constructs are used when choices are resolved by the external world. The subscript x records which branch is chosen in a worst-case input.

4.2. **Compatibility of Skeletons with Session Types.** Given an external channel $c : A$ provided by a process, suppose a user provides a skeleton K. To check the compatibility of K with A, we introduce the judgment

$$\Phi \vdash K \leqslant A, \tag{4.1}$$

where Φ is a typing context for functional-layer skeletons. The judgment (4.1) states that K is a valid skeleton of session type A, given that the external channel c is provided by a process. Figure 8 defines (4.1). Conversely, if the external channel is provided by the external world, we use the dual judgment $\Phi \vdash A \leqslant K$. Its definition is symmetric to Figure 8.

$$\boxed{\Phi \vdash K \leqslant A}$$

$$\frac{}{\Phi \vdash \mathbf{1} \leqslant \mathbf{1}}\,\text{K:Ter} \qquad \frac{\Phi \vdash H : b \qquad \Phi \vdash K \leqslant A}{\Phi \vdash H \supset K \leqslant b \supset A}\,\text{K:ValIn}$$

$$\frac{\Phi \vdash A_1 \leqslant K_1 \qquad \Phi \vdash K_2 \leqslant A_2}{\Phi \vdash K_1 \multimap K_2 \leqslant A_1 \multimap A_2}\,\text{K:ChannelIn}$$

$$\frac{\forall j \in N'.\Phi \vdash K_j \leqslant A_j \qquad \emptyset \subset N' \subseteq N}{\Phi \vdash \&_x\{\ell_i : K_i \mid i \in N'\} \leqslant \&\{\ell_j : A_j \mid j \in N\}}\,\text{K:ExtChoice}$$

Figure 8: Key rules in the compatibility relation. The judgment $\Phi \vdash H : b$ in K:ValIn means the functional-layer skeleton H has type b.

Interestingly, the relations $K \leqslant A$ and $A \leqslant K$ coincide with the subtyping relation $\leqslant$ of session types [GH05]. Upon reflection, this makes sense: because skeleton K admits some of the semantic objects of session type A, K can be considered as a subtype of A.

Due to K:Ter, $K = \mathbf{1}$ is the only skeleton compatible with session type $\mathbf{1}$. Hence, skeletons K are disallowed from stopping halfway when the corresponding session types A have not terminated yet.

This restriction eliminates the interdependence between skeletons. For instance, consider a concurrent process P with two channels c_1 and c_2. In each iteration, P either closes both c_1 and c_2 or keeps them open. If c_1 and c_2 are open, P receives one incoming message on c_1 and two incoming messages on c_2. So if a (worst-case) input on c_1 has size $n \in \mathbb{N}$, a (worst-case) input of c_2 must have size $2n$. That is, there is interdependence between the skeletons of c_1 and c_2. Thanks to K:Ter, when skeleton K_i on c_i $(i = 1, 2)$ terminates, the corresponding session type A_i for c_i must also terminate. This only happens exactly when (worst-case) inputs on c_1 and c_2 are n and $2n$, respectively, for some $n \in \mathbb{N}$.

In K:ChannelIn, the first premise uses the dual judgment. This is because, in $A_1 \multimap A_2$, the input type A_1 reverses the roles of the channel provider and client. In K:ExtChoice, skeleton K includes all labels from a non-empty subset $N' \subseteq N$. As we assume that the channel is provided by a process in the network, the choice of i in $\&\{\ell_i : A_i \mid i \in N\}$ is made by the external world. Therefore, the skeleton is allowed to limit the set of i's.

4.3. **Finite Input Portion of Skeletons.** Worst-case inputs must be finite. Otherwise, two technical challenges would arise:

(1) It is non-trivial to define a worst-case input when inputs may be infinite.
(2) Existing SMT solvers cannot solve constraints over infinite worst-case inputs.

Worst-Case Infinite Inputs. Consider a process $\cdot; 0 \vdash P :: (c : A)$, where[4]

$$A := \&\{\mathsf{first} : \mu X. \triangleleft^2 \mathsf{int} \multimap \mathsf{int} \otimes X, \mathsf{second} : \mu X. \triangleleft^1 \mathsf{int} \multimap \mathsf{int} \otimes X\}. \tag{4.2}$$

P is willing to accept two labels. If first is chosen, every iteration on channel c requires two units of potential. Otherwise, if second is chosen, every iteration only needs one unit of potential. Which scenario has a higher cost?

One possible answer is that they both have an equal cost of ∞. If we spot a recursive session type where every iteration incurs non-zero cost, then it automatically qualifies as a worst-case input, provided that the path constraint is solvable. This idea sounds too trivial.

The second possible answer is that the first branch results in a higher cost than second because the former entails two units of cost per iteration, while the latter incurs only one unit of cost per iteration. Therefore, at any moment in time, the first branch has a higher cumulative cost than the second branch.

Although this answer seems reasonable, it implicitly treats each iteration in (4.2) equally. But it is questionable whether the iterations inside the two branches in (4.2) can be treated equally. For instance, each iteration in the first branch may take twice as much time as a single iteration in the second branch. However, because SILL provides no information about timing, it is impossible to tell the cost per unit of time.

[4]Although the notation $\mu X.A_X$ is not officially in the syntax of session types (Section 3.1), we use $\mu X.A_X$ to denote an equi-recursive session type where $X = \mu X.A_X$.

Generating Infinite Data Structures. It is tricky to solve constraint satisfaction problems for data structures of infinite size. By way of example, consider a process $\cdot; 0 \vdash P :: (c : A)$, where $A := \mu X.\mathsf{bool} \multimap X$. Suppose P is implemented such that the only worst-case input is an alternating sequence of true and false. The first question is: how do we encode an infinite stream of Booleans? A sensible idea is to encode it as a function $f : \mathbb{N} \to \mathsf{bool}$, where the input is an index in the sequence and the output is the Boolean value at that index.

A path constraint for P's worst-case input is

$$\forall x \in \mathbb{N}. f(2x) = \mathsf{true} \wedge f(2x+1) = \mathsf{false}, \tag{4.3}$$

where $f : \mathbb{N} \to \mathsf{bool}$ is an uninterpreted function that encodes an infinite stream of Booleans. We seek a concrete f that satisfies the formula (4.3). Unfortunately, the current SMT technologies seem incapable of finding suitable f in this example. We ran two SMT solvers, Z3 [dMB08] and CVC4 [BCD+11], on the SMT-LIB2 encoding of (4.3) (Appendix B.2). Neither SMT solver could verify the satisfiability.

To go around these two challenges of infinite worst-case inputs, we require the input portion of a skeleton to be finite. Appendix B.2 provides details.

4.4. **Input Generation with Loose Cost Bounds.** Suppose the external world faces a choice between two branches. To resolve this choice during symbolic execution, we calculate the cost bounds of these branches and pick the one with a higher bound. How do we calculate the cost bound of a channel? We sum all resource annotations along a worst-case path on the channel's skeleton. But calculating cost bounds is non-trivial due to the intertwining of input and output across different channels.

For illustration, consider P with two external channels

$$c_1 : A_1; 0 \vdash P :: (c_2 : A_2), \tag{4.4}$$

where $A_1 := \oplus\{\mathsf{expensive} : \triangleright^2 \mathbf{1}, \mathsf{cheap} : \mathbf{1}\}$ and $A_2 := \oplus\{\mathsf{expensive} : \triangleleft^3 \mathbf{1}, \mathsf{cheap} : \mathbf{1}\}$. The external world first chooses between $\mathsf{expensive}$, which entails two units of potential, and cheap. P next chooses between $\mathsf{expensive}$ and cheap. If P chooses $\mathsf{expensive}$, three units of potential are sent as input to P.

Input and output are intertwined in this example. P first inputs a label (possibly with potential) from the external world, then outputs a label, and lastly inputs potential again. Additionally, the input and output happen on different channels: the first input happens on c_1, while the output and second input (i.e. three units of potential) happen on c_2.

According to the judgment (4.4), the total cost bound of P seems $2 + 3 = 5$. It is achieved when $\mathsf{expensive}$ is selected on both c_1 and c_2. Hence, to achieve the worst-case cost, the external world should choose $\mathsf{expensive}$ on c_1. Hopefully, P will choose $\mathsf{expensive}$ as well so that the cost bound of 5 is fulfilled.

However, P may fail to choose $\mathsf{expensive}$ on c_2. P's choice on c_2 may depend on the external world's choice on c_1 in such a way that we cannot have $\mathsf{expensive}$ on both channels. For instance, suppose P is implemented such that it sends the opposite label to whatever is received on c_1:

$$P := \mathsf{case}\ c_1\ \{\mathsf{expensive} \hookrightarrow \mathsf{tick}\ 2; c_2.\mathsf{cheap}, \mathsf{cheap} \hookrightarrow \mathsf{tick}\ 3; c_2.\mathsf{expensive}\}. \tag{4.5}$$

The tight cost bound is 3, which is lower than the bound deduced from (4.4). As a result, the worst-case input generation algorithm fails because it cannot find an execution path where the cost bound is tight.

In fact, SILL is already expressive enough to derive the tight cost bound of 3 for the example (4.5). This is evidenced by another valid typing judgment:

$$c_1 : A; 3 \vdash P :: (c_2 : A), \tag{4.6}$$

where $A := \oplus\{\mathsf{expensive} : \mathbf{1}, \mathsf{cheap} : \mathbf{1}\}$. In (4.6), all necessary potential comes from the initial constant potential 3 stored in P. Thus, typing judgments can misrepresent cost bounds, as exemplified by (4.4), even when resource-aware SILL is capable of deriving tight cost bounds.

The root cause is the following. Suppose resource annotations are scattered over a session type. When input (including the supply of potential) is interspersed with output, early input affects output, which in turn affects the later input's session type. Consequently, some combinations of inputs may be infeasible. Furthermore, if input and output reside on different channels, resource-annotated session types do not tell us how the paths on different channels are linked with each other. Therefore, even if SILL's type system can figure out the interdependence between input and output on different channels, this information is not captured by session types.

Addressing this issue is beyond this article's scope. For simplicity, when facing a choice between branches, we calculate and compare their cost bounds. The example (4.4) still respects relative completeness of worst-case input generation (Theorem 5.6) because the theorem require tight cost bounds.

5. Worst-Case Input Generation Algorithm

Suppose we are given a SILL program (P, Σ) and a collection K of skeletons for external channels. The worst-case input generation algorithm is displayed in Algorithm 1.

Algorithm 1 Worst-case input generation algorithm for a SILL program

1: **procedure** WC Input Generation$((P, \Sigma), K)$
2: Run AARA to infer resource-annotated session types of internal and external channels
3: Check that skeleton(s) K satisfies all requirements
4: Run symbolic execution while tracking potential
5: Solve the path constraint from the previous step

In line 2, we run AARA to derive resource annotations of both internal and external channels. Resource annotations will be used to keep track of potential during symbolic execution (line 4). In line 3, we check the following for skeletons of external channels:

- K is compatible with its original session type (Section 4.2).
- The input portion of K is finite (Section 4.3).

Section 5.1 describes how we track potential during symbolic execution (line 4). Section 5.2 formalizes the symbolic execution. If symbolic execution finds an execution path where the cost bound is tight, the corresponding path constraint is fed to an SMT solver in line 5. If the path constraint is solvable, we obtain a concrete worst-case input.

5.1. **Checking the Tightness of Cost Bounds.** Symbolic execution searches for an execution path where the cost bound is tight. Potential in SILL has a local nature: potential is distributed across processes and flows between them. Hence, instead of tracking the total potential, we locally track individual units of red potential (i.e. potential supplied by the external world) and blue potential (i.e. potential freed up by $\mathsf{tick}\ q$ for $q < 0$).

5.1.1. *Red Potential.* Red potential must eventually be consumed completely. To check it, we equip each process with a Boolean flag $r \in \{\mathsf{true}, \mathsf{false}\}$. $r = \mathsf{true}$ means the process contains no red potential. The flag is updated as follows:

- When (red) potential is supplied to the process by the external world, we set $r = \mathsf{false}$.
- Suppose $q > 0$ units of potential are transferred from one process (which initially has potential $p + q$ and the Boolean flag r_1) to another process (which initially has the flag r_2). The flag of the sender becomes $r_1 \vee (p = 0)$, and the flag of the recipient becomes $r_1 \wedge r_2$.

The first point clearly preserves the invariant that $r = \mathsf{true}$ if and only if the process contains no red potential. To justify the second point, let P be the sender of potential and Q be the recipient. If P's flag is already $r_1 = \mathsf{true}$ (i.e. P contains no red potential) before sending potential to Q, then P's flag should remain unchanged after sending potential. Conversely, suppose P's flag is $r_1 = \mathsf{false}$ before sending potential. If $p = 0$, it means P sends all potential it has, including red potential, to Q. Consequently, P no longer has any red potential left. Hence, P's flag is updated to $r = \mathsf{true}$ if $p = 0$. The update rule of Q's flag is justified by similar reasoning.

In addition, we forbid processes from throwing away potential when $r = \mathsf{false}$. Potential is thrown away by the rule relax in the type system (Figure 7). If it happens, it means the cost bound is not tight. In such an event, symbolic execution backtracks and explores another execution path.

Likewise, red potential is not allowed to flow back to the external world, since cost bounds only factor in incoming potential from the external world.

5.1.2. *Blue Potential.* Blue potential must be consumed if its generation precedes the consumption of red potential. What does it mean that one event precedes another? It is determined by the happened-before relation [Lam78], which is a well-established notion in distributed and concurrent computing.

Definition 5.1 (Happened-before relation [Lam78])**.** The happened-before relation $\rightarrow$ is the smallest binary relation closed under the following three conditions. First, if A happens before B on the same process, $A \rightarrow B$ holds. Second, if A is an event of sending a message and B is the event of receiving the message, $A \rightarrow B$ holds. Third, if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

We now explain how to check whether blue potential, if generated before red potential is consumed, is also consumed entirely. Suppose that, during symbolic generation, blue potential is generated by $\mathsf{tick}\ q$, where $q < 0$, in process P. We assign a fresh ID, say $\mathrm{blue}_i \in \mathcal{C}$, to this newly created blue potential. Here, $\mathcal{C} = \{\mathrm{red}\} \cup \{\mathrm{blue}_i \mid i \in \mathbb{N}\}$ is the set of all IDs for red and blue potential.

Definition 5.1 defines what it means for one process to be synchronized with another process.

Definition 5.1 (Synchronization of two processes)**.** *We say that process Q is synchronized with P since some event of P if and only if Q's current state happens after the event.*

For concreteness, suppose that P has generated blue potential of the ID blue_1. During symbolic execution, we track the following:

- The set of processes that have been synchronized (Definition 5.1) with P since the generation of blue_1. We track such processes by passing around the ID blue_1 whenever a message is sent by P or other processes that carry blue_1. At any moment, the set of synchronized processes is given by the set of processes carrying the ID blue_1.
- What other potential is consumed by a process synchronized with P.
- Whether blue_1 is completely consumed.

To understand why we track these items, assume (i) Q has been synchronized with P, (ii) Q consumes red potential, and (iii) blue_1 is not entirely consumed by the end of program execution. Because blue_1 is generated before red is consumed, we can substitute blue potential for red potential. Hence, we can lower the amount of necessary red potential without plunging into negative potential. That is, the cost bound is not tight in this case.

Moreover, there are other possibilities where we can substitute blue potential for red potential. Consider the following scenario. We want to substitute certain blue potential (with ID, say, blue_1) for red potential, where blue_1 was generated before the consumption of red potential. However, blue_1 is entirely consumed. So we must find another blue potential (say blue_2) that can substitute for blue_1. This is possible when (i) blue_2 is generated before blue_1's consumption and (ii) blue_2 is not entirely consumed. Figure 9 depicts this situation.

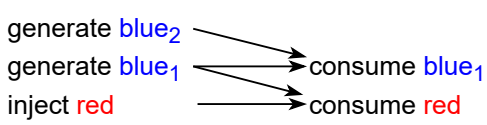

Figure 9: Alternating chain of the happened-before relation between events. "inject red" refers to an event of injecting red potential to some process from the external world. "generate blue" refers to the generation of blue potential.

If blue_2 is not completely used, we can push the leftover potential of blue_2, from "generate blue_2" to "inject red," along the alternating path of $\rightarrow$'s in Figure 9. As a result, we can reduce the total red potential injected to the program, thereby lowering the cost bound.

To correctly detect such alternating paths of $\rightarrow$, symbolic execution maintains a graph whose set of nodes is $\mathcal{C}$ (i.e. set of IDs for red and blue potential). The graph has an edge (v_1, v_2) if and only if potential v_2 is consumed after potential v_1 is generated. In this graph, if there is a path from blue_i to red such that blue_i is not entirely consumed, then the cost bound is not tight. In such an event, symbolic execution backtracks and explores another execution path.

Theorem 5.2 states the correctness of tracking potential.

Theorem 5.2 (Checking tightness of cost bounds)**.** *If red and blue potential is tracked without encountering issues, then the cost bound is tight.*

5.2. **Symbolic Execution.** In symbolic execution, we run a SILL program on a skeleton while keeping track of potential. Because the operational semantics of SILL is given by a multiset rewriting system, we also use it to define symbolic execution.

Symbolic execution involves two types of predicates:

$$\mathsf{proc}(\Delta; q \vdash P :: (c : A), \phi, \mathrm{IDs}) \quad \mathsf{msg}(c, M, \phi, \mathrm{IDs}).$$

The left predicate represents a well-typed process $\Delta; q \vdash P :: (c : A)$, where A is either a resource-annotated skeleton (if c is an external channel) or a resource-annotated session type (if c is an internal channel). Logical formula ϕ is a path constraint so far and will later be fed to an SMT solver. $\mathrm{IDs} = (\mathrm{IDs}_s, \mathrm{IDs}_p)$ is a pair of finite sets of potential's IDs. The first set $\mathrm{IDs}_s \subset \{\mathrm{blue}_i \mid i \in \mathbb{N}\}$ tracks synchronization: if process P's IDs_s contains blue_i, then P's current state happens after blue_i was generated. The second set $\mathrm{IDs}_p \subset \mathcal{C}$ tracks potential transfer: if process P's IDs_p contains an ID i, then P contains the (red or blue) potential identified by i.

$\mathsf{msg}(c, M, \phi, \mathrm{IDs})$ represents a message M (encoded as as process) that provides channel c. Logical formula ϕ is a path constraint carried by the message.

Figure 10 displays key rewriting rules. Remaining rules are given in Appendix C.4.

$$\frac{\mathsf{proc}(\Delta, c : b \supset A; p \vdash \mathsf{send}\ c\ e; P :: (d : D), \phi_1, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad e \Downarrow \langle \phi_2, v \rangle \qquad c' \text{ is fresh}}{\mathsf{proc}(\Delta, c' : A; p \vdash P[c'/c] :: (d : D), \phi_1, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad \mathsf{msg}(c', \mathsf{send}\ c\ v; c' \leftarrow c, \phi_2, (\mathrm{IDs}_s, \emptyset))} \supset S$$

$$\frac{\mathsf{proc}(\Delta; p \vdash \mathsf{case}\ c\ \{\ell_i \hookrightarrow P_i \mid i \in N\} :: (c : \&_x\{\ell_i : A_i \mid i \in N\}), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p))}{\mathsf{proc}(\Delta; p \vdash P_k[c'/c] :: (c' : A_k), \phi \wedge (x = k), (\mathrm{IDs}_s, \mathrm{IDs}_p))} \&R_{\text{external}}$$

$$\frac{\mathsf{proc}(\cdot; p \vdash \mathsf{close}\ c :: (c : \mathbf{1}), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad \mathrm{red} \notin \mathrm{IDs}_p}{\mathsf{msg}(c, \mathsf{close}\ c, \phi, (\mathrm{IDs}_s, \emptyset))} \mathbf{1}S$$

$$\frac{\mathsf{proc}(\Delta, c : \triangleleft^q A; p + q \vdash \mathsf{pay}\ c\ \{q\}; P :: (d : B), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad c' \text{ is fresh} \qquad \mathrm{red} \notin \mathrm{IDs}_p}{\mathsf{proc}(\Delta, c' : A; p \vdash P[c'/c] :: (d : B), \phi, (\mathrm{IDs}_s, (p = 0)\ ?\ \emptyset : \mathrm{IDs}_p))} \triangleleft L_{\text{external}}$$

$$\frac{\mathsf{proc}(\Delta; p \vdash \mathsf{get}\ c\ \{q\}; P :: (c : \triangleleft^q A), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p))}{\mathsf{proc}(\Delta; p + q \vdash P :: (c : A), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p \cup \{\mathrm{red}\}))} \triangleleft R_{\text{external}}$$

$$\frac{\mathsf{proc}(\Delta; p + q \vdash \mathsf{tick}\ q; P :: (c : A), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p))}{\mathsf{proc}(\Delta; p \vdash P :: (c : A), \phi, (\mathrm{IDs}_s, (p = 0)\ ?\ \emptyset : \mathrm{IDs}_p))} \mathsf{tick}_{>0}$$

$$\frac{\mathsf{proc}(\Delta; p \vdash \mathsf{tick}\ (-q); P :: (c : A), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad \mathrm{blue}_i \in \mathcal{C} \text{ is fresh}}{\mathsf{proc}(\Delta; p + q \vdash P :: (c : A), \phi, (\mathrm{IDs}_s \cup \{\mathrm{blue}_i\}, \mathrm{IDs}_p \cup \{\mathrm{blue}_i\}))} \mathsf{tick}_{<0}$$

Figure 10: Key rules in the process-layer symbolic execution. Throughout the rules, we have $q > 0$. $e \Downarrow \langle \phi, v \rangle$ means e evaluates to a (symbolic) value v with a path constraint ϕ. A ternary operator $(b)\ ?\ e_1 : e_2$ returns e_1 if b evaluates to true and e_2 otherwise.

In the rule $\supset S$, when a message is sent, it also carries the sender's IDs_s. It is then added to the recipient's IDs_s.

The rule $\&R_{\text{external}}$ resolves an external choice on an external channel. A label $k \in N$ is randomly chosen among those labels with the highest cost bound. The path constraint ϕ is then augmented with a constraint $x = k$, indicating that the external world should choose $k \in N$ to trigger worst-case performance. If we find out later that the current path's cost bound is not tight, we backtrack and try a different $k' \neq k$.

The rule $\mathbf{1}_S$ forbids red potential from being wasted: red potential must not remain when a process terminates. Likewise, $\triangleleft L_{\text{external}}$, which is for an external channel, forbids red potential from flowing back to the external world. Furthermore, if we waste blue potential (i.e. $\text{IDs}_p \setminus \{\text{red}\} \neq \emptyset$) in $\mathbf{1}_S$ and $\triangleleft L_{\text{external}}$, we must record its IDs because we do not want to waste blue potential that could have been substituted for red potential.

In $\triangleleft R_{\text{external}}$, the process receives red potential from the external world. So the ID red is added to the recipient's IDs_p.

Finally, we have two rules for tick. In $\mathsf{tick}_{>0}$, the potential stored in the process is consumed. Whenever this rule is applied, we must record the pair $(\text{IDs}_s, \text{IDs}_p)$. This pair indicates which blue potential ID was generated before red potential is consumed. In $\mathsf{tick}_{<0}$, blue potential is generated. Hence, we generate a fresh ID and add it to IDs_s and IDs_p.

5.3. **Soundness and Relative Completeness.** Theorems 5.2 assures us that symbolic execution's high-level idea is sound. However, it assumes that the symbolic execution algorithm correctly tracks costs and potential. To prove this assumption, we show a simulation between the symbolic execution and cost semantics. The simulation then leads to the soundness and relative completeness of our worst-case input generation algorithm.

Definition 5.3 defines a similarity relation between predicates.

Definition 5.3 (Similarity between predicates)**.** Fix S to be a solution (i.e. a mapping from skeleton variables to concrete values) to a path constraint generated by symbolic execution. The similarity relation $\sim$ between a predicate in symbolic execution and a predicate in the cost semantics is defined by

$$\frac{S \vdash P_{\text{sym}} = P_{\text{cost}}}{\mathsf{proc}(_; _ \vdash P_{\text{sym}} :: (c : _), _, _) \sim \mathsf{proc}(c, P_{\text{cost}})} \qquad \frac{S \vdash M_{\text{sym}} = M_{\text{cost}}}{\mathsf{msg}(c, M_{\text{sym}}, _) \sim \mathsf{msg}(c, M_{\text{cost}})}.$$

Here, $S \vdash P_{\text{sym}} = P_{\text{cost}}$ means P_{sym} and P_{cost} are equal under the mapping S.

Definition 5.3 can be extended from predicates to configurations. Proposition 5.4 establishes a simulation between the symbolic execution and cost semantics.

Proposition 5.4 (Simulation for soundness)**.** *Suppose we are given three configurations: $C_{1,sym}$, $C_{2,sym}$, and $C_{1,cost}$. The first two configurations are used in the symbolic execution, and the last one is used in cost semantics. These configurations satisfy two conditions: (i) $C_{1,sym}$ transitions to $C_{2,sym}$ in one step of symbolic execution and (ii) $C_{1,sym} \sim C_{1,cost}$ holds. Then there exists a configuration $C_{2,cost}$ of cost semantics such that the following diagram commutes:*

$$\begin{array}{ccc} C_{1,sym} & \xrightarrow[w]{} & C_{2,sym} \\ \wr & & \wr \\ C_{1,cost} & \xrightarrow[w]{\leq 1} & C_{2,cost} \end{array}$$

In this diagram, $C_{1,sym} \underset{w}{\rightarrow} C_{2,sym}$ means $\lceil \bowtie \rceil(C_{1,sym}) - \lceil \bowtie \rceil(C_{2,sym}) = w$, where $\lceil \bowtie \rceil(\cdot)$ denotes a cost bound of a configuration (see Definition C.3). Likewise, $C_{1,cost} \underset{w}{\rightarrow} C_{2,cost}$

means $C_{1,cost}$ transitions to $C_{2,cost}$ such that the net cost increases by w. The arrow $\rightarrow^{\leq 1}$ means the number of steps is either zero or one.

In Proposition 5.4, one transition step in symbolic execution may correspond to zero steps in cost semantics. It happens when we transfer potential by rewriting rules such as $\triangleleft L_{\text{external}}$ in symbolic execution—they do not have corresponding rules in the cost semantics.

Finally, Theorem 5.5 states the soundness of the worst-case input generation algorithm, and Theorem 5.6 states the relative completeness of the algorithm. All proofs are in Appendix C.3.

Theorem 5.5 (Soundness of worst-case input generation)**.** *Given a collection $K_1, \ldots, K_n$ of skeletons, suppose the symbolic execution algorithm successfully terminates. Let ϕ be a path constraint generated by symbolic execution and $t \in [\![K_1, \ldots, K_n]\!]$ be an input satisfying ϕ. Then t has the highest high-water-mark cost of all inputs from $[\![K_1, \ldots, K_n]\!]$.*

Theorem 5.6 (Relative completeness of worst-case input generation)**.** *Given a SILL program and a collection K of skeletons, assume the following:*

- *The processes are typable in resource-aware SILL.*
- *The cost bound of K is tight.*
- *Symbolic execution terminates.*
- *The background theory for path constraints is decidable.*

Then the algorithm returns a valid worst-case input.

6. Case Study: Web Server and Browsers

We model the interaction between a web server and multiple web browsers. For each browser, a new channel is spawned, and the browser first engages in a three-way handshake protocol with the server (as in TCP). Once the handshake protocol is successfully completed, the browser and server proceed to the main communication phase for data transfer.

This case study considers the non-monotone resource metric of memory. Suppose the server requires (i) one memory cell for the handshake protocol and (ii) another memory cell for the subsequent communication after the handshake. The first memory cell stores so-called sequence numbers that are established during the handshake. The second memory cell for the main communication phase stores data about a browser.

Let c be a channel provided by the server and used by browsers. Without loss of generality, suppose we have two browsers that want to communicate with the server. We examine two implementations of the server. In the first implementation (Section 6.1), the two browsers' sessions run independently of each other. So the server cannot control how the sessions are scheduled. By contrast, in the second implementation (Section 6.2), the server coordinates sessions with the help of a scheduler.

6.1. Independent Sessions.

In the first implementation, the type of the provided channel is $c : A \otimes A \otimes \mathbf{1}$. That is, the server sequentially spawns a new channel of type A for each of the two browsers. The server after spawning two channels is depicted in Figure 11 (a). The providers P of these channels run independently of each other—the server cannot control the order of events on these channels.

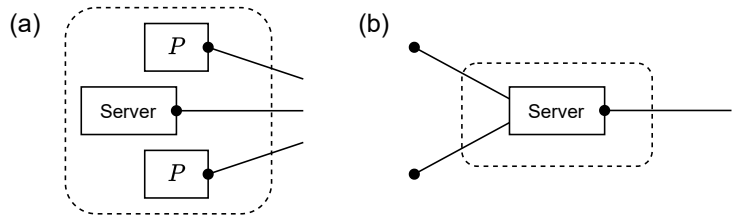

Figure 11: (a) First model of a web server. Here, an independent process P is assigned to each browser's session that wants to communicate with the server. (b) Second model of a web server where a scheduler is modeled.

Resource-annotated session type A is defined as

$$A := \mathsf{int} \supset \triangleleft^1(\mathsf{int} \times \mathsf{int}) \wedge \&\{\mathsf{ack} : \mathsf{int} \supset \oplus\{\mathsf{success} : \triangleleft^1\mathbf{1}, \mathsf{failure} : \mathbf{1}\}, \mathsf{timeout} : \mathbf{1}\}.$$

In A, the browser first sends an integer x, and the server sends back two integers: $1 + x$ and y. In response, the browser sends either label ack followed by an integer (ideally $1 + y$) or label timeout to indicate that the browser is inactive.

After successful completion of the handshake, the server sends success and terminates. If the browser sends back a wrong integer (i.e. an integer different from $1 + y$) to the server, the server sends failure. After success and failure, there is no further communication between the server and browser. This is for simplicity in our modeling. In practice, the success branch has further communication.

What skeleton should we use? The session type $A \otimes A \otimes \mathbf{1}$ of c cannot be immediately used as a skeleton, because its type-level paths have different cost bounds. It is clear that the highest cost bound arises when the external world chooses ack instead of timeout. Therefore, the overall skeleton for the external channel c is $K_1 \otimes K_2 \otimes \mathbf{1}$, where K_i is

$$K_i := z_{i,1} \supset \triangleleft^1(\mathsf{int} \times \mathsf{int}) \wedge \&\{\mathsf{ack} : z_{i,2} \supset \oplus\{\mathsf{success} : \triangleleft^1\mathbf{1}, \mathsf{failure} : \mathbf{1}\}\}.$$

Here, $z_{i,1}$ and $z_{i,2}$ are skeleton variables.

The implementation of such a server is given in D.1.

The cost bound of $K \otimes K \otimes \mathbf{1}$ is 4 because each instance of K has a cost bound of 2. The worst-case input generation algorithm successfully generates a worst-case input

$$z_{1,1} = x_1 \quad z_{1,2} = y_1 + 1 \quad z_{2,1} = x_2 \quad z_{2,2} = y_2 + 1, \tag{6.1}$$

where $x_1, x_2 \in \mathbb{Z}$ are unconstrained, and y_1, y_2 are the integers sent back from the server to a browser. Constants y_1, y_2 are assumed to be hard-coded in the server's code.

More generally, if we have n many browsers, the cost bound becomes $2n$. It is tight: because different sessions run independently, in the worst case, we will need $2n$ memory cells at the peak memory usage. As before, the worst-case input generation algorithm can compute a correct worst-case input of high-water-mark cost $2n$. It suggests that adversaries can overwhelm the server's memory by deploying a large number of browsers.

This vulnerability is reminiscent of a denial-of-service (DoS) attack. However, our implementation does not quite model the true DoS attack. A DoS attack creates a large number of so-called half-open sessions (i.e. the completion of the handshake is delayed by withholding ack). On the other hand, in our model, the worst-case performance arises when a large number of sessions are completed (rather than staying half-open) and run concurrently.

To faithfully model a DoS attack, we would need to model the passage of time and the exact scheduling of sessions, both of which are challenging in SILL.

6.2. **Coordinating Sessions with Schedulers.** Now consider an alternative implementation where it is the external world that spawns channels. We have $c : A \multimap A \multimap \mathbf{1}$, where A (without resource annotations) is

$$A := \mathsf{int} \wedge (\mathsf{int} \times \mathsf{int}) \supset \oplus\{\mathsf{ack} : \mathsf{int} \wedge \&\{\mathsf{success} : \mathbf{1}, \mathsf{failure} : \mathbf{1}\}, \mathsf{timeout} : \mathbf{1}\}. \tag{6.2}$$

A's resource annotation depends on how the browsers' sessions are scheduled.

Unlike in Section 6.1, in this section, the two channels are directly connected to the server (Figure 11 (b)). Hence, the server can/must coordinate the communication sessions on the channels. For example, the server can use a round-robin scheduler that alternates between the two browsers. Another possibility is a sequential scheduler: the server serves the first browser and then moves on to the second one after the first browser is finished.

With a round-robin scheduler, we obtain the same high-water mark as Section 6.1. With a sequential scheduler, the resource-annotated type of channel c is $A_{\text{anno}} \multimap A \multimap \mathbf{1}$, where

$$A_{\text{anno}} := \mathsf{int} \wedge \triangleright^1(\mathsf{int} \times \mathsf{int}) \supset \oplus\{\mathsf{ack} : \mathsf{int} \wedge \&\{\mathsf{success} : \triangleright^1\mathbf{1}, \mathsf{failure} : \mathbf{1}\}, \mathsf{timeout} : \mathbf{1}\}.$$

and A is given in (6.2). It is a valid typing judgment because once the server finishes talking with the first browser, two memory cells are freed up and are reused for the second browser. Therefore, the sequential scheduler's cost bound is lower than the round-robin scheduler's.

More generally, if we have n browsers, the sequential scheduler's cost bound remains 2. Further, thanks to the soundness of resource-annotated session types, the bound 2 is a valid cost bound. Therefore, adversaries cannot overwhelm the server by sending a large number of communication requests.

7. Related Work

Resource Analysis. Resource analysis of programs aims to derive symbolic cost bounds. Numerous approaches exist: type systems [CW00, Vas08, Dan08, LG11, ADL17, ÇBG+17, HVH19], recurrence relations [Weg75, Gro01, AAG+07, KCBR17, KMLD19, CLD20], term rewriting [AM13, BEF+14, HM14, MS20], and static analysis [GMC09, ADLM15, CFG19]. Among type-based approaches is AARA. Linear AARA was first developed by Hofmann and Jost [HJ03] and later extended to univariate polynomial bounds [HH10], multivariate polynomial bounds [HAH12], and exponential bounds [KH20]. AARA has also been incorporated into imperative programming [CHS15], parallel programming [HS15], and probabilistic programming [NCH18, WKH20, AMS20].

Session Types. Session types, which describe communication protocols on channels, were originally proposed by Honda [Hon93]. Caires and Pfenning [CP10] build a session type system whose logical counterpart is intuitionistic linear logic. Their session type system has been integrated into a functional programming language using contextual monads, resulting in the language SILL [TCP13, PG15]. Resource-aware SILL [DHP18] incorporates linear AARA into SILL (excluding shared channels). Nomos [DBHP19] is a session-typed programming language that uses resource-aware SILL to infer gas bounds of smart contracts. Wadler has developed a session type system based on classical linear logic [Wad12]. Binary session types have also been extended to multiparty ones [HYC08].

Worst-Case Input Generation. The present work was inspired by the type-guided worst-case input generation for functional programming by Wang and Hoffmann [WH19]. While [WH19] focuses on monotone resource metrics in sequential programming, the present work considers more general non-monotone resource metrics in message-passing concurrent programming. Non-monotone resource metrics, when combined with concurrency of processes, pose challenges to worst-case input generation.

WISE [BJS09] is the first work to use symbolic execution for worst-case input generation. It first explores an entire search space for a small input to identify a worst-case execution path. This path is then generalized to a branch policy to handle larger inputs. The use of branch policies reduces the search space of large inputs, thereby making worst-case input generation more scalable. SPF-WCA [LKP17] extends WISE with path policies that take into account histories when we determine which branch to take during symbolic execution.

Instead of branch and path policies, Wang and Hoffmann [WH19] and we use resource-annotated types to guide symbolic execution. One advantage of type-guided symbolic execution is that worst-case input generation becomes sound. Another advantage is that we learn how many non-monotone resources can be transferred between concurrent processes.

The fuzzing research community has investigated worst-case input generation. SlowFuzz [PZKJ17] is the first fuzzer that automatically finds worst-case inputs with gray-box access to programs. PerfFuzz [LPSS18] extends SlowFuzz with multi-dimensional objectives. MemLock [WWL+20] focuses on memory consumption bugs. Although fuzzing generally offers neither soundness nor relative completeness, it is more scalable than static-analysis-based worst-case input generation because fuzzers do not analyze programs' internal workings.

8. Conclusion

It is non-trivial to generate worst-case inputs to concurrent programs under non-monotone resource metrics. The high-water-mark cost of a concurrent program depends on how processes are scheduled at runtime. As a result, haphazardly executing a concurrent program may not reveal its correct high-water-mark cost.

In this work, we have developed the first sound worst-case input generation algorithm for message-passing concurrent programming under non-monotone resource metrics. The key insight is to have resource-annotated session types guide symbolic execution. We have also identified several technical challenges posed by session types in the design of skeletons. We have proved the soundness and relative completeness of our algorithm. Finally, we have presented a simple case study of a web server's memory usage, illustrating the utility of the worst-case input generation algorithm.

References

[AAG+07] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 157–172, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-71316-6_12.

[ADL17] Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110287.

[ADLM15] Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 152–164, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2784731.2784753.

[AM13] Martin Avanzini and Georg Moser. A combination framework for complexity. In *24th International Conference on Rewriting Techniques and Applications (RTA'13)*, 2013.

[AMS20] Martin Avanzini, Georg Moser, and Michael Schaper. A modular cost analysis for probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):172:1–172:30, November 2020. doi:10.1145/3428240.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BEF+14] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In *20th Int. Conf. on Tools and Alg. for the Constr. and Anal. of Systems (TACAS'14)*, 2014.

[BJS09] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *2009 IEEE 31st International Conference on Software Engineering*, pages 463–473, 2009. doi:10.1109/ICSE.2009.5070545.

[BP17] Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110281.

[BTP19] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In Luís Caires, editor, *Programming Languages and Systems*, pages 611–639, Cham, 2019. Springer International Publishing.

[ÇBG+17] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 316–329, New York, NY, USA, January 2017. Association for Computing Machinery. doi:10.1145/3009837.3009858.

[CFG19] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Non-polynomial Worst-Case Analysis of Recursive Programs. *ACM Transactions on Programming Languages and Systems*, 41(4):20:1–20:52, October 2019. doi:10.1145/3339984.

[CHS15] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 467–478, New York, NY, USA, June 2015. Association for Computing Machinery. doi:10.1145/2737924.2737955.

[CLD20] Joseph W. Cutler, Daniel R. Licata, and Norman Danner. Denotational recurrence extraction for amortized analysis. *Proceedings of the ACM on Programming Languages*, 4(ICFP):97:1–97:29, August 2020. doi:10.1145/3408979.

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory*, CONCUR'10, page 222–236, Berlin, Heidelberg, 2010. Springer-Verlag.

[CS06] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Electronic Notes in Theoretical Computer Science*, 165:145–176, 2006. Proceedings of the 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006). URL: https://www.sciencedirect.com/science/article/pii/S1571066106005202, doi:10.1016/j.entcs.2006.05.043.

[CW00] Karl Crary and Stephnie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 184–198, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/325694.325716.

[Dan08] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 133–144, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1328438.1328457.

[DBHP19] Ankush Das, Stephanie Balzer, Jan Hoffmann, and Frank Pfenning. Resource-aware session types for digital contracts. *CoRR*, abs/1902.06056, 2019. URL: http://arxiv.org/abs/1902.06056, arXiv:1902.06056.

[DHP18] Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*,

LICS '18, page 305–314, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3209108.3209146.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[GH05] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 11 2005. Copyright - Springer-Verlag 2005; Last updated - 2014-08-02; CODEN - AINFA2. URL: https://search-proquest-com.cmu.idm.oclc.org/scholarly-journals/subtyping-session-types-pi-calculus/docview/275047514/se-2?accountid=9902.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 127–139, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1480881.1480898.

[Gro01] Bernd Grobauer. Cost recurrences for DML programs. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 253–264, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/507635.507666.

[HAH12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource Aware ML. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 781–786, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-31424-7_64.

[HH10] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential. In Andrew D. Gordon, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 287–306, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-11957-6_16.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 185–197, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/604131.604148.

[HM14] Martin Hofmann and Georg Moser. Amortised resource analysis and typed polynomial interpretations. In *Rewriting and Typed Lambda Calculi (RTA-TLCA;14)*, 2014.

[Hon93] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, pages 509–523, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[HS15] Jan Hoffmann and Zhong Shao. Automatic Static Cost Analysis for Parallel Programs. In Jan Vitek, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 132–157, Berlin, Heidelberg, 2015. Springer. doi:10.1007/978-3-662-46669-8_6.

[HVH19] Martin A. T. Handley, Niki Vazou, and Graham Hutton. Liquidate your assets: Reasoning about resource usage in liquid Haskell. *Proceedings of the ACM on Programming Languages*, 4(POPL):24:1–24:27, December 2019. doi:10.1145/3371092.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, January 2008. doi:10.1145/1328897.1328472.

[KCBR17] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. Non-linear reasoning for invariant synthesis. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158142.

[KH20] David M. Kahn and Jan Hoffmann. Exponential Automatic Amortized Resource Analysis. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 359–380, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-45231-5_19.

[KMLD19] G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. Recurrence extraction for functional programs through call-by-push-value. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371083.

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984.*, pages 558–565, July 1978. 2000 PODC Influential Paper Award (later renamed the Edsger W. Dijkstra Prize in Distributed Computing). Also awarded an ACM SIGOPS Hall

of Fame Award in 2007. URL: https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/.

[LG11] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, 2011.

[LKP17] Kasper Luckow, Rody Kersten, and Corina Păsăreanu. Symbolic complexity analysis using context-preserving histories. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 58–68, 2017. doi:10.1109/ICST.2017.13.

[LPSS18] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 254–265, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3213846.3213874.

[MS20] Georg Moser and Manuel Schneckenreither. Automated amortised resource analysis for term rewrite systems. *Science of Computer Programming*, 185:102306, January 2020. doi:10.1016/j.scico.2019.102306.

[NCH18] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 496–512, New York, NY, USA, June 2018. Association for Computing Machinery. doi:10.1145/3192366.3192394.

[PG15] Frank Pfenning and Dennis Griffith. Polarized substructural session types. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015. doi:10.1007/978-3-662-46678-0_1.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, Massachusetts, 2002.

[PS09] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 101–110, 2009. doi:10.1109/LICS.2009.8.

[PZKJ17] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2155–2168, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3134073.

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Matrix Analysis and Applications*, 6(2):306–13, April 1985.

[TCP13] Bernardo Toninho, Luis Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, page 350–369, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-37036-6_20.

[Vas08] Pedro B. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, University of St Andrews, UK, 2008.

[Wad12] Philip Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 273–286, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2364527.2364568.

[Weg75] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, September 1975. doi:10.1145/361002.361016.

[WH19] Di Wang and Jan Hoffmann. Type-guided worst-case input generation. *Proceedings of the ACM on Programming Languages*, 3(POPL):13:1–13:30, January 2019. doi:10.1145/3290326.

[WKH20] Di Wang, David M. Kahn, and Jan Hoffmann. Raising expectations: Automating expected cost analysis with types. *Proceedings of the ACM on Programming Languages*, 4(ICFP):110:1–110:31, August 2020. doi:10.1145/3408992.

[WWL+20] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page

765–777, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377811.3380396.

Appendix A. Resource-Aware SILL

A.1. **Cost Semantics and the Type System.** Figure 6 already gives some rewriting rules of the cost semantics of SILL. Remaining rules are displayed in Figure 12.

$$\frac{\mathsf{proc}(d, w, c \leftarrow e \leftarrow \overline{c_i}; Q_c) \qquad e \Downarrow (x \leftarrow P_{x,\overline{x_i}} \leftarrow \overline{x_i}) \qquad c' \text{ is fresh}}{\mathsf{proc}(c', 0, P_{c',\overline{c_i}}) \qquad \mathsf{proc}(d, w, Q_{c'})}\mathsf{spawn}$$

$$\frac{\mathsf{msg}(c, w_1, \mathsf{send}\ c\ v; c \leftarrow c') \qquad \mathsf{proc}(d, w_2, x \leftarrow \mathsf{recv}\ c; P_x)}{\mathsf{proc}(d, w_1 + w_2, P_v[c'/c])}\wedge R$$

$$\frac{\mathsf{proc}(c, w, \mathsf{send}\ c\ e; P) \qquad e \Downarrow v \qquad c' \text{ is fresh}}{\mathsf{proc}(c', w, P[c'/c]) \qquad \mathsf{msg}(c, 0, \mathsf{send}\ c\ v; c \leftarrow c')}\wedge S$$

$$\frac{\mathsf{proc}(d, w, \mathsf{send}\ c_1\ c_2; P) \qquad c_1' \text{ is fresh}}{\mathsf{proc}(d, w, P[c_1'/c_1]) \qquad \mathsf{msg}(c_1', 0, \mathsf{send}\ c_1\ c_2; c_1' \leftarrow c_1)}\multimap S \qquad \frac{\mathsf{msg}(c_1', w_1, \mathsf{send}\ c_1\ c_2; c_1' \leftarrow c_1) \quad \mathsf{proc}(c_1, w_2, x \leftarrow \mathsf{recv}\ c_1; P_x)}{\mathsf{proc}(c_1, w_1 + w_2, P_{c_2}[c_1'/c_1])}\multimap R$$

$$\frac{\mathsf{msg}(c_1, w_1, \mathsf{send}\ c_1\ c_2; c_1 \leftarrow c_1') \quad \mathsf{proc}(d, w_2, x \leftarrow \mathsf{recv}\ c_1; P_x)}{\mathsf{proc}(d, w_1 + w_2, P_{c_2}[c_1'/c_1])}\otimes R \qquad \frac{\mathsf{proc}(c_1, w, \mathsf{send}\ c_1\ c_2; P) \qquad c_1' \text{ is fresh}}{\mathsf{proc}(c_1', w, P[c_1'/c_1]) \qquad \mathsf{msg}(c_1, 0, \mathsf{send}\ c_1\ c_2; c_1 \leftarrow c_1')}\otimes S$$

$$\frac{\mathsf{proc}(d, w, c.\ell_k; P) \qquad c' \text{ is fresh}}{\mathsf{proc}(d, w, P[c'/c]) \qquad \mathsf{msg}(c', 0, c.\ell_k; c' \leftarrow c)}\&S \qquad \frac{\mathsf{msg}(c', w_1, c.\ell_k; c' \leftarrow c) \quad \mathsf{proc}(c, w_2, \mathsf{case}\ c\ \{\overline{\ell_i \hookrightarrow P_i}\})}{\mathsf{proc}(c, w_1 + w_2, P_k[c'/c])}\&R$$

$$\frac{\mathsf{msg}(c, w_1, c.\ell_k; c \leftarrow c') \quad \mathsf{proc}(d, w_2, \mathsf{case}\ c\ \{\overline{\ell_i \hookrightarrow P_i}\})}{\mathsf{proc}(d, w_1 + w_2, P_k[c'/c])}\oplus R \qquad \frac{\mathsf{proc}(c, w, c.\ell_k; P) \qquad c' \text{ is fresh}}{\mathsf{proc}(c', w, P[c'/c]) \qquad \mathsf{msg}(c, 0, c.\ell_k; c \leftarrow c')}\oplus S$$

$$\frac{\mathsf{proc}(c_1, w, c_1 \leftarrow c_2)}{\mathsf{msg}(c_1, w, c_1 \leftarrow c_2)}\mathsf{fwd}_s \qquad \frac{\mathsf{proc}(c_2, w_1, P) \qquad \mathsf{msg}(c_1, w_2, c_1 \leftarrow c_2)}{\mathsf{proc}(c_1, w_1 + w_2, P[c_1/c_2])}\mathsf{fwd}_r^+$$

$$\frac{\mathsf{msg}(c_1, w_1, c_1 \leftarrow c_2) \quad \mathsf{proc}(d, w_2, P)}{\mathsf{proc}(d, w_1 + w_2, P[c_2/c_1])}\mathsf{fwd}_r^- \qquad \frac{\mathsf{proc}(c, w, \mathsf{close}\ c)}{\mathsf{msg}(c, w, \mathsf{close}\ c)}\mathbf{1}S \qquad \frac{\mathsf{msg}(c, w_1, \mathsf{close}\ c) \quad \mathsf{proc}(d, w_2, \mathsf{wait}\ c; P)}{\mathsf{proc}(d, w_1 + w_2, P)}\mathbf{1}R$$

Figure 12: Remaining rules in the cost semantics of SILL. The judgment $e \Downarrow v$ means functional term e evaluates to value v.

To treat $\mathsf{proc}(\cdot)$ and $\mathsf{msg}(\cdot)$ uniformly, message M in predicate $\mathsf{msg}(c, M)$ is encoded as a process [DHP18], just like P in predicate $\mathsf{proc}(c, P)$. The grammar of message M, which

is subsumed by the grammar of processes, is presented below.

$$\begin{aligned} M ::= {} & c \leftarrow c' \mid c.\ell_i; c \leftarrow c' \mid c.\ell_i; c' \leftarrow c \\ & \mid \mathsf{send}\ c\ v; c \leftarrow c' \mid \mathsf{send}\ c\ v; c' \leftarrow c \\ & \mid \mathsf{send}\ c_1\ c_2; c_1 \leftarrow c_1' \mid \mathsf{send}\ c_1\ c_2; c_1' \leftarrow c_1 \mid \mathsf{close}\ c. \end{aligned}$$

Here, c and c' are channels, v is a functional-layer value, and $q \in \mathbb{Q}_{>0}$ is a quantity of potential.

Some rules of the type system of resource-aware SILL are already given in Figure 6. The remaining rules are presented in Figure 13.

Appendix B. Session Skeletons

B.1. **Compatibility of Skeletons with Session Types.** The key rules of the compatibility relation $\Gamma \vdash K \leqslant A$ are already given in Figure 8. The remaining rules are in Figure 14. The dual $\Gamma \vdash A \leqslant K$ is defined similarly, so its formal definition is omitted.

The definition of compatibility can be extended to infinite session types by taking the greatest fixed point (Chapter 21 of [Pie02]). We omit formal treatment of infinite session types so that the readers can focus on the core ideas.

B.2. **Checking Finiteness of Input Portions.** The SMT-LIB2 encoding of (4.3) is displayed below.

```
(set-logic UFLIA)
(declare-fun f (Int) Bool)
(assert (forall ((x Int))
    (and (= (f (* 2 x)) true)
         (= (f (+ (* 2 x) 1)) false))))
(check-sat)
```

How do we check the finiteness of an input portion? Our approach is to count the number of input actions at the type level and check if the number is infinite. Given a skeleton K, the judgment

$$\Gamma \vdash K\ \mathsf{countInput}(n) \tag{B.1}$$

states that K contains at most n many input actions at the type level, where $n \in \mathbb{N} \cup \{\infty\}$. A context Γ maps type variables (i.e. X in $\mu X.K_X$) to their associated numbers of input actions. Here, input actions are defined from the viewpoint of the channel's provider. We also have the dual judgment $\Gamma \vdash K\ \mathsf{countOutput}(n)$ stating that K contains n output actions at the type level.

The judgment (B.1) is defined in Figures 15. The dual judgment is defined similarly, so we omit its definition. To compute the number of input and output actions, we construct a template derivation tree according to this inference system. In the tree, we use variables n's to record the number of input/output actions, and collect constraints on them. Finally, we solve them by an LP solver.

$$\boxed{\Phi; \Delta; q \vdash P :: (c : A)}$$

$$\frac{\Phi \vdash e : \{x : A \leftarrow \overline{x_i : A_i} \mathbin{@} p\} \qquad \Delta_1 = \{\overline{c_i : A_i}\} \qquad \Phi; \Delta_2, c : A; q \vdash Q_c :: (d : D)}{\Phi; \Delta_1, \Delta_2; p + q \vdash (c \leftarrow e \leftarrow \overline{c_i}; Q_c) :: (d : D)}\,\mathsf{spawn}$$

$$\wedge L\ \frac{\Phi, x : b; \Delta, c : A; p \vdash P_x :: (d : D)}{\Phi; \Delta, c : b \wedge A; p \vdash x \leftarrow \mathsf{recv}\ c; P_x :: (d : D)} \qquad \wedge R\ \frac{\Phi; \Delta; p \vdash P :: (x : A) \qquad \Phi \vdash e : b}{\Phi; \Delta; p \vdash \mathsf{send}\ c\ e; P :: (c : b \wedge A)}$$

$$\frac{\Phi; \Delta, c_1 : A_2; p \vdash P :: (d : D)}{\Phi; \Delta, c_2 : A_1, c_1 : A_1 \multimap A_2; p \vdash \mathsf{send}\ c_1\ c_2; P :: (d : D)}\multimap L \qquad \frac{\Phi; \Delta, x : A_1; p \vdash P_x :: (c : A_2)}{\Phi; \Delta; p \vdash (x \leftarrow \mathsf{recv}\ c; P_x) :: (c : A_1 \multimap A_2)}\multimap R$$

$$\frac{\Phi; \Delta, x : A_1, c : A_2; p \vdash P_x :: (d : D)}{\Phi; \Delta, c : A_1 \otimes A_2; p \vdash x \leftarrow \mathsf{recv}\ c; P_x :: (d : D)}\otimes L \qquad \frac{\Phi; \Delta; p \vdash P :: (c_1 : A_2)}{\Phi; \Delta, c_2 : A_1; p \vdash \mathsf{send}\ c_1\ c_2; P :: (c_1 : A_1 \otimes A_2)}\otimes R$$

$$\frac{\Phi; \Delta, c : A_k; p \vdash P :: (d : D)}{\Phi; \Delta, c : \&\{\ell_i : A_i\}; p \vdash c.\ell_k; P : (d : D)}\&L \qquad \frac{\forall i. \Phi; \Delta; p \vdash P_i :: (c : A_i)}{\Phi; \Delta; p \vdash \mathsf{case}\ c\ \{\overline{\ell_i \hookrightarrow P_i}\} :: (c : \&\{\overline{\ell_i : A_i}\})}\&R$$

$$\frac{\forall i. \Phi; \Delta, c : A_i; p \vdash P_i :: (d : D)}{\Phi; \Delta, c : \oplus\{\overline{\ell_i : A_i}\}; p \vdash \mathsf{case}\ c\ \{\overline{\ell_i \hookrightarrow P_i}\} :: (d : D)}\oplus L \qquad \frac{\Phi; \Delta; p \vdash P :: (c : A_k)}{\Phi; \Delta; q \vdash (c.\ell_k; P) :: (c : \oplus\{\overline{\ell_i : A_i}\})}\oplus R$$

$$\frac{}{\Phi; c_2 : A; 0 \vdash c_1 \leftarrow c_2 :: (c_1 : A)}\mathsf{fwd} \qquad \frac{\Phi; \Delta; p \vdash P :: (d : D)}{\Phi; \Delta, c : \mathbf{1}; p \vdash \mathsf{wait}\ c; P :: (d : D)}\mathbf{1}L$$

$$\frac{}{\Phi; \cdot; 0 \vdash \mathsf{close}\ c :: (c : \mathbf{1})}\mathbf{1}R \qquad \frac{\Phi; \Delta, c : A; p \vdash P :: (d : D)}{\Phi; \Delta, c : \triangleleft^q A; p + q \vdash \mathsf{pay}\ c\ \{q\}; P :: (d : D)}\triangleleft L$$

$$\frac{\Phi; \Delta; p + q \vdash P :: (c : A)}{\Phi; \Delta; p \vdash \mathsf{get}\ c\ \{q\}; P :: (c : \triangleleft^q A)}\triangleleft R \qquad \frac{\Phi; \Phi; \Delta, c : A; p + q \vdash P :: (d : D)}{\Phi; \Phi; \Delta, c : \triangleright^q A; p \vdash \mathsf{get}\ c\ \{q\}; P :: (d : D)}\triangleright L$$

$$\frac{\Phi; \Delta; p \vdash P :: (c : A)}{\Phi; \Delta; p + q \vdash \mathsf{pay}\ c\ \{q\}; P :: (c : \triangleright^q A)}\triangleright R$$

Figure 13: Remaining rules of the type system of resource-aware SILL. $\Phi \vdash e : \tau$ in the rules spawn and $\supset L$ is a typing judgment for the functional layer.

B.3. **Extraction of Cost Bounds.** A type-level path of a session type (or a skeleton) is a path on the session type where all choices of labels (including external and internal choices) are resolved.

$$\boxed{\Phi; \Delta \vdash K \leqslant A}$$

$$\frac{\Phi; \Delta \vdash K \leqslant A}{\Phi; \Delta \vdash b \wedge K \leqslant b \wedge A}\text{K:ValOut} \qquad \frac{\Phi; \Delta \vdash K_1 \leqslant A_1 \qquad \Phi; \Delta \vdash K_2 \leqslant A_2}{\Phi; \Delta \vdash K_1 \otimes K_2 \leqslant A_1 \otimes A_2}\text{K:ChannelOut}$$

$$\frac{\forall i \in N. \Phi; \Delta \vdash K_i \leqslant A_i}{\Phi; \Delta \vdash \oplus\{\ell_i : K_i \mid N\} \leqslant \oplus\{\ell_i : A_i \mid N\}}\text{K:InChoice} \qquad \frac{\Phi; \Delta \vdash K \leqslant A}{\Phi; \Delta \vdash \triangleleft^q K \leqslant \triangleleft^q A}\text{K:Get}$$

$$\frac{\Phi; \Delta \vdash K \leqslant A}{\Phi; \Delta \vdash \triangleright^q K \leqslant \triangleright^q A}\text{K:Pay}$$

Figure 14: Remaining rules in the compatibility relation $\Phi; \Delta \vdash K \leqslant A$.

$$\boxed{\Gamma \vdash K\ \mathsf{countInput}(n)}$$

$$\frac{}{\Gamma \vdash \mathbf{1}\ \mathsf{countInput}(0)} \qquad \frac{\Gamma \vdash K_2\ \mathsf{countInput}(n)}{\Gamma \vdash H \supset K\ \mathsf{countInput}(n+1)} \qquad \frac{\Gamma \vdash K\ \mathsf{countInput}(n)}{\Gamma \vdash b \wedge K\ \mathsf{countInput}(n)}$$

$$\frac{\Gamma \vdash K_1\ \mathsf{countOutput}(n_1) \qquad \Gamma \vdash K_2\ \mathsf{countInput}(n_2)}{\Gamma \vdash K_1 \multimap K_2\ \mathsf{countInput}(n_1 + n_2 + 1)}$$

$$\frac{\Gamma \vdash K_1\ \mathsf{countInput}(n_1) \qquad \Gamma \vdash K_2\ \mathsf{countInput}(n_2)}{\Gamma \vdash K_1 \otimes K_2\ \mathsf{countInput}(n_1 + n_2)}$$

$$\frac{\forall i \in N. \Gamma \vdash K_i\ \mathsf{countInput}(n_i)}{\Gamma \vdash \&_x\{\ell_i : K_i \mid i \in N\}\ \mathsf{countInput}(1 + \max_{i \in N}\{n_i\})} \qquad \frac{\Gamma \vdash K\ \mathsf{countInput}(n)}{\Gamma \vdash \triangleleft^q K\ \mathsf{countInput}(n+1)}$$

$$\frac{\forall i \in N. \Gamma \vdash K_i\ \mathsf{countInput}(n_i)}{\Gamma \vdash \oplus\{\ell_i : K_i \mid i \in N\}\ \mathsf{countInput}(\max_{i \in N}\{n_i\})} \qquad \frac{\Gamma \vdash K\ \mathsf{countInput}(n)}{\Gamma \vdash \triangleright^q K\ \mathsf{countInput}(n)}$$

Figure 15: Number of input actions in a skeleton.

Definition B.1 (Type-level paths)**.** *Given a session type/skeleton A, its set of type-level paths is*

$$\begin{aligned}
\mathsf{path}(A_1 \multimap A_2) &:= \{p_1 \multimap p_2 \mid p_i \in \mathsf{path}(A_i)\} \\
\mathsf{path}(A_1 \otimes A_2) &:= \{p_1 \otimes p_2 \mid p_i \in \mathsf{path}(A_i)\} \\
\mathsf{path}(\&\{\ell_i : A_i \mid i \in N\}) &:= \{\&\{\ell_k : p_k\} \mid k \in N, p_k \in \mathsf{path}(A_k)\} \\
\mathsf{path}(\oplus\{\ell_i : A_i \mid i \in N\}) &:= \{\oplus\{\ell_k : p_k\} \mid k \in N, p_k \in \mathsf{path}(A_k)\} \\
\mathsf{path}(\mathbf{1}) &:= \{\mathbf{1}\}.
\end{aligned}$$

The sets of type-level paths for other type constructors can be defined straightforwardly.

The issue illustrated by (4.5) arises from the interactive nature of resource-aware SILL. A process in concurrent programming receives incoming messages (i.e. input) and sends outgoing messages (i.e. output) that may depend on the previous input. Afterwards, the process repeats the process of receiving input and sending output. In this way, input and output are intertwined in SILL. As a consequence of the interdependence between input

$$\lceil\triangleleft\rceil(H \supset K) := \lceil\triangleleft\rceil(K) \qquad \lceil\triangleleft\rceil(b \wedge K) := \lceil\triangleleft\rceil(K)$$
$$\lceil\triangleleft\rceil(K_1 \multimap K_2) := \lceil\triangleright\rceil(K_1) + \lceil\triangleleft\rceil(K_2) \qquad \lceil\triangleleft\rceil(K_1 \otimes K_2) := \lceil\triangleleft\rceil(K_1) + \lceil\triangleleft\rceil(K_2)$$
$$\lceil\triangleleft\rceil(\&_x\{\ell_i : K_i \mid i \in N\}) := \max_{i\in N}\lceil\triangleleft\rceil(K_i) \qquad \lceil\triangleleft\rceil(\oplus\{\ell_i : K_i \mid i \in N\}) := \max_{i\in N}\lceil\triangleleft\rceil(K_i)$$
$$\lceil\triangleleft\rceil(\triangleleft^q K) := q + \lceil\triangleleft\rceil(K) \qquad \lceil\triangleleft\rceil(\triangleright^q K) := \lceil\triangleleft\rceil(K)$$
$$\lceil\triangleleft\rceil(\mathbf{1}) := 0.$$

Figure 16: Cost bounds of skeletons.

and output across different channels, not all combinations of type-level paths on different channels are feasible. Furthermore, if some type-level paths have higher cost bounds than others but are infeasible, it becomes challenging to figure out which type-level paths to explore during worst-case input generation.

The cost bound of a skeleton is defined below.

Definition B.2 (Cost bounds of skeletons)**.** *Given a skeleton K for an external channel provided by a process, its cost bound, denoted by $\lceil\triangleleft\rceil(K)$, is defined in Figure 16. The dual cost bound $\lceil\triangleright\rceil(\cdot)$, which is used in the definition of $\lceil\triangleleft\rceil(K_1 \multimap K_2)$, is defined analogously.*

In order for Definition B.2 to make sense, K's input portion must be finite.

$\triangleleft^q$ contributes to cost bounds in Definition B.2, but $\triangleright^q$ does not. If cost bounds factored in outgoing potential as well as incoming potential, cost bounds might depend on the output size. In skeletons, while the input size is fixed, the output size is not known statically. Consequently, before looking for an execution path with a tight cost bound, the worst-case input generation algorithm would first need to maximize the cost bound (by minimizing the output size). As this will complicate the algorithm, we do not factor outgoing potential into cost bounds.

Appendix C. Worst-Case Input Generation

C.1. **Problem Statement.** An input to a SILL program is a collection of predicates $\mathsf{msg}(\cdot,\cdot)$ from the external world. To formalize inputs in SILL, we fix a naming scheme for channels in the rewriting system. As of now, whenever a fresh channel name is needed, the rewriting system does not specify the fresh name. Consequently, when we formally define an input as a collection of predicates $\mathsf{msg}(c, M)$, it is unclear what precisely c should be. Thus, to formally define inputs, we must determine fresh channel names deterministically. Although it is possible to devise a desirable naming scheme for channels, due to its complexity, we omit its formal definition. Throughout this section, we adopt such a naming scheme.

Each input is a collection of predicates $\mathsf{msg}(c, M)$. Given a channel $c : K$ (where K is a skeleton), let $\mathrm{inputs}(K, c)$ denote the set of possible inputs from the perspective of c's provider. It is defined in Figure 17. The dual $\mathrm{outputs}(K, c)$ is the set of all outputs for c's provider. Since $\mathrm{outputs}(\cdot,\cdot)$ is defined similarly, we omit its definition.

Definition C.1 (Worst-case inputs)**.** Consider a network of processes with well-typed external channels $c_1 : K_1, \ldots, c_m : K_m, c_{m+1} : K_{m+1}, \ldots, c_n : K_n$. Here, $c_1, \ldots, c_m$ are provided by processes inside the network, and $c_{m+1}, \ldots, c_n$ are provided by the external

$$
\begin{aligned}
\text{inputs}(H \supset K, c) &:= \{t \cup \{\mathsf{msg}(c, \mathsf{send}\ c\ v)\} \mid v \in [\![H]\!], t \in \text{inputs}(K, c')\} \\
\text{inputs}(b \wedge K, c) &:= \text{inputs}(K, c) \\
\text{inputs}(K_1 \multimap K_2, c) &:= \{t_1 \cup t_2 \cup \{\mathsf{msg}(c', \mathsf{send}\ c\ d; c' \leftarrow c)\} \mid \\
&\qquad c', d \text{ are fresh}, t_1 \in \text{inputs}(K_2, c'), t_2 \in \text{outputs}(K_1, d)\} \\
\text{inputs}(K_1 \otimes K_2, c) &:= \{t_1 \cup t_2 \mid c', d \text{ are fresh}, t_1 \in \text{inputs}(K_2, c'), t_2 \in \text{inputs}(K_1, d)\} \\
\text{inputs}(\&\{\ell_i : K_i \mid i \in N\}, c) &:= \{t \cup \{\mathsf{msg}(c', c.\ell_k; c' \leftarrow c)\} \mid j \in N, c' \text{ is fresh}, t \in \text{inputs}(K_j, c')\} \\
\text{inputs}(\oplus\{\ell_i : K_i \mid i \in N\}, c) &:= \{t \mid j \in N, c' \text{ is fresh}, t \in \text{inputs}(K_j, c')\} \\
\text{inputs}(\mathbf{1}, c) &:= \emptyset
\end{aligned}
$$

Figure 17: Set of possible inputs of a skeleton. The dual $\text{outputs}(K, c)$ is the set of all outputs for c's provider.

world. $K_1, \ldots, K_m$ are skeletons compatible with their original session types. The set of all possible inputs to this network is given by

$$[\![K_1, \ldots, K_n]\!] := \prod_{1 \le i \le m} \text{inputs}(K_i, c_i) \times \prod_{m < i \le n} \text{outputs}(K_i, c_i).$$

A worst-case input $(t_1, \ldots, t_n) \in [\![K_1, \ldots, K_n]\!]$ is such that, if we add $\bigcup_{1 \le i \le n} t_i$ to the initial configuration of the network and run it, we obtain the highest high-water-mark cost of all possible inputs from $[\![K_1, \ldots, K_n]\!]$.

C.2. **Checking Tightness of Cost Bounds.**

Proposition C.2 (Checking depletion of red potential)**.** *Suppose red potential is tracked as described above without encountering the following issues:*

- *Red potential remains in some process at the end of symbolic execution;*
- *Red potential is thrown away or flows back to the external world.*

Then the red potential supplied to the program is completely consumed.

Proof. Whenever potential flows from one process to another, we assume that red potential (if there is any) in the sender is split evenly, and half of it is transferred. Throughout symbolic execution, the flag r is true if and only if the current process contains red potential. This invariant is proved by case analysis. Firstly, when potential is supplied by the external world, the Boolean flag of the recipient is set to false, and it is consistent with the invariant. Secondly, when potential is transferred, the Boolean flag is correctly updated in a way that preserves the invariant. Lastly, red potential is never discarded. Thanks to the Boolean flag's invariant, when symbolic execution successfully finishes, red potential should be completely gone because processes terminate only when $r = \mathsf{true}$ (i.e. red potential is absent). Therefore, if symbolic execution successfully terminates, red potential will have been consumed completely.

We split red potential evenly when potential is transferred to another process. Even if red potential is split differently, it does not affect our conclusion. For example, suppose we split red potential such that it stays in the sender or it all goes to the recipient. Then the set of processes with red potential is a subset of what we will have when red potential is split evenly. When symbolic execution terminates, if red potential is divided evenly, the set

of processes with red potential is empty. Hence, even if we change the way red potential is split, by the end of symbolic execution, red potential must be completely gone.

In conclusion, regardless of how we split red potential, when symbolic execution successfully terminates, all red potential is gone. □

THEOREM 5.2. *If red and blue potential is tracked without encountering issues, then the cost bound is tight.*

Proof. A cost bound is equal to the amount of (red) potential supplied by the external world to a SILL program. It follows from Proposition C.2 that red potential is entirely consumed if symbolic execution successfully terminates. Also, because symbolic execution properly tracks blue potential, there should be no path from unconsumed blue potential to red potential in Figure 9.

To show the tightness of a cost bound, it suffices to explicitly construct a schedule of processes whose high-water mark is equal to the cost bound. Firstly, suppose we only have one process P. If red potential is consumed completely, then when the last red potential is consumed, the high-water mark of P is equal to the total red potential supplied by the external world. This can be seen from Figure 2 (b).

Next, consider a non-trivial case where we have two processes: P_1 and P_2. Let t_1 be the moment in P_1's timeline when it completely consumes all potential in Figure 9, including red potential. Define t_2 similarly for P_2. We can then run P_1 and P_2 until they stop exactly at t_1 and t_2, respectively. For example, if P_1 sends a message and P_2 receives it before t_2, then t_1 must happen after the event of P_1 sending the message. Because potential is completely consumed at t_2, any potential generated before t_2, including potential on P_1 right before sending the message, must be gone before P_1 reaches t_1. Furthermore, the high-water mark when t_1 and t_2 are reached is equal to the total red potential supplied by the external world. No potential is captured by messages in transit; otherwise, it would contradict the assumption that all potential in Figure 9 is entirely consumed by the time t_1 and t_2. Thus, all potential in Figure 9, including red potential, should have been consumed by P_1 and P_2 when they reach t_1 and t_2. Therefore, the net cost at this point is equal to the total red potential supplied by the external world.

This reasoning can be generalized to more than two processes. □

C.3. **Soundness and Relative Completeness.**

Definition C.3 (Cost bounds of configurations)**.** Suppose C is a configuration with external channels $c_1, \ldots, c_m, c_{m+1}, \ldots, c_n$. Here, $c_1, \ldots, c_m$ are provided by processes in C, whereas $c_{m+1}, \ldots, c_n$ are provided by the external world. Let $K_1, \ldots, K_n$ be the skeletons of external channels. Also, let p be the total potential locally stored in processes in C. The cost bound of configuration C is defined as

$$\lceil \bowtie \rceil(C) := p + \sum_{i=1}^{m} \lceil \triangleright \rceil(K_i) + \sum_{i=m+1}^{n} \lceil \triangleleft \rceil(K_i).$$

$\lceil \triangleright \rceil(K_i)$ is the cost bound of skeleton K_i when the external channel is provided by some process in the network (Definition B.2). The dual is $\lceil \triangleleft \rceil(K_i)$.

Definition C.4 (Similarity between configurations)**.** Fix S to be a solution to a path constraint generated by symbolic execution. Consider a configuration C_{sym} for the symbolic

execution and a configuration C_{cost} for cost semantics. Let t be an input induced by S. The similarity relation $C_{\text{sym}} \sim C_{\text{cost}}$ holds if and only if there is an injection from the predicates of $C_{\text{sym}} \cup t$ to those of C_{cost} such that each pair in the injection satisfies the similarity relation (Definition 5.3).

All proofs related to the soundness and relative completeness of worst-case input generation are presented in this section.

THEOREM 5.4. *Suppose we are given three configurations: $C_{1,sym}$, $C_{2,sym}$, and $C_{1,cost}$. The first two configurations are used in the symbolic execution, and the last one is used in cost semantics. These configurations satisfy two conditions: (i) $C_{1,sym}$ transitions to $C_{2,sym}$ in one step of symbolic execution and (ii) $C_{1,sym} \sim C_{1,cost}$ holds. Then there exists a configuration $C_{2,cost}$ of cost semantics such that the following diagram commutes:*

$$\begin{tikzcd}
C_{1,sym} \arrow[r, "w"'] \arrow[d, squiggly, dash] & C_{2,sym} \arrow[d, squiggly, dash] \\
C_{1,cost} \arrow[r, "w"', "\leq 1"] & C_{2,cost}
\end{tikzcd}$$

In this diagram, $C_{1,sym} \underset{w}{\rightarrow} C_{2,sym}$ means $\lceil\bowtie\rceil(C_{1,sym}) - \lceil\bowtie\rceil(C_{2,sym}) = w$, where $\lceil\bowtie\rceil(\cdot)$ denotes a cost bound of a configuration (see Definition C.3). Likewise, $C_{1,cost} \underset{w}{\rightarrow} C_{2,cost}$ means $C_{1,cost}$ transitions to $C_{2,cost}$ such that the net cost increases by w. The arrow $\rightarrow^{\leq 1}$ means the number of steps is either zero or one.

Proof. By case analysis on the rewriting rules of symbolic execution. Strictly speaking, in the rules for termination and forwarding in symbolic execution, potential may be discarded. Therefore, the above commutative diagram is not quite correct: after one transition step in both symbolic execution and the cost semantics, the potential may decrease by w, while the net cost stays the same. However, this can be fixed by saving all potential, including the one that is actually discarded before termination and forwarding in symbolic execution. □

THEOREM 5.5. *Given a collection $K_1, \ldots, K_n$ of skeletons, suppose the symbolic execution algorithm successfully terminates. Let ϕ be a path constraint generated by symbolic execution and $t \in [\![K_1, \ldots, K_n]\!]$ be an input satisfying ϕ. Then t has the highest high-water-mark cost of all inputs from $[\![K_1, \ldots, K_n]\!]$.*

Proof. Proposition 5.4 shows that symbolic execution correctly keeps track of potential and net cost: both of them change by the same amount (but in the opposite direction). Furthermore, by Theorem 5.2, the cost bound is tight for t. Therefore, the high-water-mark cost of t is the highest. □

Proposition C.5 (Simulation for completeness)**.** *Suppose we are given three configurations: $C_{1,cost}$, $C_{2,cost}$, and $C_{1,sym}$. The first two configurations are used in the cost semantics, and the last one is used in symbolic execution. These configurations satisfy two conditions: (i) $C_{1,cost}$ transitions to $C_{2,cost}$ in one step of cost semantics and (ii) $C_{1,sym} \sim C_{1,cost}$ holds. Then there exists a configuration $C_{2,sym}$ of symbolic execution such that the following diagram*

commutes:

$$\begin{array}{ccc} C_{1,sym} & \xrightarrow[w]{\geq 1} & C_{2,sym} \\ \wr & & \wr \\ C_{1,cost} & \xrightarrow[w]{} & C_{2,cost} \end{array}$$

In this diagram, $C_{1,cost} \underset{w}{\rightarrow} C_{2,cost}$ *means* $C_{1,cost}$ *transitions to* $C_{2,cost}$ *such that the net cost increases by* w*. The arrow* $\rightarrow^{\geq 1}$ *means the number of steps is at least one. Likewise,* $C_{1,sym} \underset{w}{\rightarrow} C_{2,sym}$ *means* $\lceil \bowtie \rceil(C_{1,sym}) - \lceil \bowtie \rceil(C_{2,sym}) = w$*.*

Proof. By case analysis on the rewriting rules of the cost semantics (Figure 6). In symbolic execution, when processes terminate or forward, the processes are sometimes allowed to throw away potential. As a result, this breaks the above commutative diagram because potential may decreases while the net cost stays the same. To work around this issue, as done in the proof of Proposition 5.4, we save potential somewhere instead of throwing it away. □

THEOREM 5.6. *Given a SILL program and a collection* K *of skeletons, assume the following:*

- *The processes are typable in resource-aware SILL.*
- *The cost bound of* K *is tight.*
- *Symbolic execution terminates.*
- *The background theory for path constraints is decidable.*

Then the algorithm returns a valid worst-case input.

Proof. Proposition 5.4 states that any transition in the cost semantics can be simulated by symbolic execution. Also, we can straightforwardly prove the dual of Theorem 5.2: if there exists a finite worst-case execution path π whose high-water-mark cost matches the cost bound, then we can find π by symbolic execution. Combining these two results yields the relative completeness of the worst-case input generation algorithm under the above assumptions. □

C.4. **Symbolic Execution.** Some of the key rules for symbolic execution are already presented in Figure 10. The remaining key rules are given in Figures 18 and 19. Figures 10, 18, and 19 cover half of all rules. The other half is just the dual of the three figures in this article; hence, it is omitted.

In the symbolic execution for the process layer, we need to transfer potential. Hence, we augment the grammar of message M (Appendix A.1) as follows:

$$M ::= \cdots \mid \mathsf{pay}\ c\ \{q\}; c \leftarrow c' \mid \mathsf{pay}\ c\ \{q\}; c' \leftarrow c.$$

Here, $q \in \mathbb{Q}_{>0}$ denotes the quantity of potential to be transferred.

Skeleton variables of functional types are added to path constraints during the functional layer's symbolic execution. For instance, suppose a process's code contains $\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. During symbolic execution, if we choose to explore the first branch, we add b to a path constraint. As symbolic execution for the functional layer is already presented in a prior work [WH19], this article omits it.

$$\frac{\begin{array}{c}\mathsf{proc}(\Delta_1, \Delta_2; p + q \vdash (c \leftarrow e \leftarrow \overline{c_i}; Q_c) :: (d : D), \phi_2, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \\ e \Downarrow \langle \phi_1, x \leftarrow P_{x,\overline{x_i}} \leftarrow \overline{x_i} \rangle \qquad c' \text{ is fresh}\end{array}}{\begin{array}{c}\mathsf{proc}(\Delta_1; p \vdash P_{c',\overline{c_i}} :: (c' : A), \phi_1, (\mathrm{IDs}_s, (p = 0)\ ?\ \emptyset : \mathrm{IDs}_p)) \\ \mathsf{proc}(\Delta_2, c' : A; q \vdash Q_{c'} :: (d : D), \phi_2, (\mathrm{IDs}_s, (q = 0)\ ?\ \emptyset : \mathrm{IDs}_p))\end{array}}\ \mathsf{spawn}$$

$$\frac{\begin{array}{c}\mathsf{msg}(c', \mathsf{send}\ c\ v; c' \leftarrow c, \phi_1, (\mathrm{IDs}_{s,1}, \emptyset)) \\ \mathsf{proc}(\Delta; p \vdash x \leftarrow \mathsf{recv}\ c; P_x :: (c : b \supset A), \phi_2, (\mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,2}))\end{array}}{\mathsf{proc}(\Delta; p \vdash P_v[c'/c] :: (c : A), \phi_1 \wedge \phi_2, (\mathrm{IDs}_{s,1} \cup \mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,2}))}\ \supset R_{\text{internal}}$$

$$\frac{\mathsf{proc}(\Delta; p \vdash x \leftarrow \mathsf{recv}\ c; P_x :: (c : H \supset K), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p))}{\mathsf{proc}(\Delta; p \vdash P_H[c'/c] :: (c : K), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p))}\ \supset R_{\text{external}}$$

$$\frac{\mathsf{proc}(\Delta, c_1 : A_1 \multimap A_2, c_2 : A_1; p \vdash \mathsf{send}\ c_1\ c_2; P :: (d : D), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad c_1' \text{ is fresh}}{\mathsf{proc}(\Delta, c_1' : A_2; p \vdash P[c_1'/c_1] :: (d : D), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad \mathsf{msg}(c_1', \mathsf{send}\ c_1\ c_2; c_1' \leftarrow c_1, \top, (\mathrm{IDs}_s, \emptyset))}\ \multimap S$$

$$\frac{\begin{array}{c}\mathsf{msg}(c_1', \mathsf{send}\ c_1\ c_2; c_1' \leftarrow c_1, \top, (\mathrm{IDs}_{s,1}, \emptyset)) \\ \mathsf{proc}(\Delta; p \vdash x \leftarrow \mathsf{recv}\ c_1; P_x :: (c_1 :: A_1 \multimap A_2), \phi, (\mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,2}))\end{array}}{\mathsf{proc}(\Delta; p \vdash P_{c_2}[c_1'/c_1] :: (c_1' : A_2), \phi, (\mathrm{IDs}_{s,1} \cup \mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,2}))}\ \multimap R_{\text{internal}}$$

$$\frac{\mathsf{proc}(\Delta; p \vdash x \leftarrow \mathsf{recv}\ c_1; P_x :: (c_1 :: A_1 \multimap A_2), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad c_1' \text{ is fresh}}{\mathsf{proc}(\Delta; p \vdash P_{c_2}[c_1'/c_1] :: (c_1' : A_2), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p))}\ \multimap R_{\text{external}}$$

$$\frac{\mathsf{proc}(\Delta, c : \&\{\ell_i : A_i \mid i \in N\}; p \vdash c.\ell_k; P :: (d : D), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad c' \text{ is fresh}}{\mathsf{proc}(\Delta, c' : A_k; p \vdash P[c'/c] :: (d : D), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad \mathsf{msg}(c', c.\ell_k; c' \leftarrow c, \top, (\mathrm{IDs}_s, \emptyset))}\ \&S$$

$$\frac{\begin{array}{c}\mathsf{msg}(c', c.\ell_k; c' \leftarrow c, \top, (\mathrm{IDs}_{s,1}, \emptyset)) \\ \mathsf{proc}(\Delta; p \vdash \mathsf{case}\ c\ \{\ell_i \hookrightarrow P_i \mid i \in N\} :: (c : \&\{\ell_i : A_i \mid i \in N\}), \phi, (\mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,2}))\end{array}}{\mathsf{proc}(\Delta; p \vdash P_k[c'/c] :: (c' : A_k), \phi, (\mathrm{IDs}_{s,1} \cup \mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,2}))}\ \&R_{\text{internal}}$$

Figure 18: Remaining key rules in the symbolic execution for the process layer (part 1). The judgment $e \Downarrow \langle \phi, v \rangle$ means e evaluates to a (symbolic) value v with a path constraint ϕ.

Appendix D. Case Study: Server and Browsers

The implementation of a server with independent sessions (Section 6.1) is

$$d_1 \leftarrow P \leftarrow \cdot; \mathsf{send}\ c\ d_1; d_2 \leftarrow P \leftarrow \cdot; \mathsf{send}\ c\ d_2; \mathsf{close}\ c,$$

$$\frac{\mathsf{proc}(c_2 : A; p \vdash c_1 \leftarrow c_2 :: (c_1 : A), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad \mathrm{red} \notin \mathrm{IDs}_p}{\mathsf{msg}(c_1, c_1 \leftarrow c_2, \phi)}\mathsf{fwd}_s$$

$$\frac{\mathsf{proc}(\Delta; p \vdash P :: (c_2 : A), \phi_1, (\mathrm{IDs}_{s,1}, \mathrm{IDs}_{p,1})) \qquad \mathsf{msg}(c_1, c_1 \leftarrow c_2, \phi_2, (\mathrm{IDs}_{s,2}, \emptyset))}{\mathsf{proc}(\Delta; p \vdash P[c_1/c_2] :: (c_1 : A), \phi_1 \wedge \phi_2, (\mathrm{IDs}_{s,1} \cup \mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,1}))}\mathsf{fwd}_r^+$$

$$\frac{\mathsf{msg}(c_1, c_1 \leftarrow c_2, \phi_1, (\mathrm{IDs}_{s,1}, \emptyset)) \qquad \mathsf{proc}(\Delta, c_1 : A; p \vdash P :: (d : D), \phi_2, (\mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,2}))}{\mathsf{proc}(\Delta, c_2 : A; p \vdash P :: (d : D), \phi_1 \wedge \phi_2, (\mathrm{IDs}_{s,1} \cup \mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,2}))}\mathsf{fwd}_r^-$$

$$\frac{\mathsf{msg}(c, \mathsf{close}\ c, \phi_1, (\mathrm{IDs}_{s,1}, \emptyset)) \qquad \mathsf{proc}(\Delta, c : \mathbf{1}; p \vdash \mathsf{wait}\ c; P :: (d : D), \phi_2, (\mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,2}))}{\mathsf{proc}(\Delta; p \vdash P :: (d : D), \phi_1 \wedge \phi_2, (\mathrm{IDs}_{s,1} \cup \mathrm{IDs}_{s,2}, \mathrm{IDs}_p))}\mathbf{1}R_{\mathrm{internal}}$$

$$\frac{\mathsf{proc}(\Delta, c : \mathbf{1}; p \vdash \mathsf{wait}\ c; P :: (d : D), \phi_2, (\mathrm{IDs}_s, \mathrm{IDs}_p))}{\mathsf{proc}(\Delta; p \vdash P :: (d : D), \phi_1 \wedge \phi_2, (\mathrm{IDs}_s, \mathrm{IDs}_p))}\mathbf{1}R_{\mathrm{external}}$$

$$\frac{\begin{array}{c}\mathsf{proc}(\Delta, c : \triangleleft^q A; p + q \vdash \mathsf{pay}\ c\ \{q\}; P :: (d : B), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \qquad c' \text{ is fresh}\end{array}}{\begin{array}{c}\mathsf{proc}(\Delta, c' : A; p \vdash P[c'/c] :: (d : B), \phi, (\mathrm{IDs}_s, (p = 0)\ ?\ \emptyset : \mathrm{IDs}_p)) \\ \mathsf{msg}(c', \mathsf{pay}\ c\ \{p\}; c' \leftarrow c, (\mathrm{IDs}_s, \mathrm{IDs}_p))\end{array}}\triangleleft L_{\mathrm{internal}}$$

$$\frac{\begin{array}{c}\mathsf{proc}(\Delta; q \vdash \mathsf{get}\ c\ \{p\}; P :: (c : \triangleleft^p A), \phi, (\mathrm{IDs}_{s,1}, \mathrm{IDs}_{p,1})) \\ \mathsf{msg}(c', \mathsf{pay}\ c\ \{p\}; c \leftarrow c', (\mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,2}))\end{array}}{\mathsf{proc}(\Delta; p + q \vdash P[c'/c] :: (c' : A), \phi, (\mathrm{IDs}_{s,1} \cup \mathrm{IDs}_{s,2}, \mathrm{IDs}_{p,1} \cup \mathrm{IDs}_{p,2}))}\triangleleft R_{\mathrm{internal}}$$

Figure 19: Remaining key rules in the symbolic execution for the process layer (part 2).

where the process $d \leftarrow P \leftarrow \cdot$ is implemented as

$$\begin{aligned}
P := {} & x \leftarrow \mathsf{recv}\ d; \mathsf{tick}\ 1; \mathsf{send}\ d\ \langle x + 1, y\rangle; \\
& \mathsf{case}\ d\ \{\mathsf{ack} \hookrightarrow z \leftarrow \mathsf{recv}\ d; \\
& \qquad \mathsf{if}\ (z = 1 + y)\ \mathsf{then} \\
& \qquad\quad d.\mathsf{success}; \mathsf{tick}\ 1; \mathsf{tick}\ -2; \mathsf{close}\ d \\
& \qquad \mathsf{else} \\
& \qquad\quad d.\mathsf{failure}; \mathsf{close}\ d, \\
& \quad \mathsf{timeout} \hookrightarrow \mathsf{tick}\ -1; \mathsf{close}\ d\}.
\end{aligned} \tag{D.1}$$

Integers x and y are called sequence numbers and are stored in the server. This is why we have tick 1. Additionally, once the handshake is completed successfully, we run tick 1 because we assume that one memory cell is required for the subsequent communication phase. Lastly, before a channel is closed, we free up all memory.

With a sequential scheduler, P is implemented as

$$\begin{aligned}
P := {} & x_1 \leftarrow \mathsf{recv}\ d_1; \mathsf{tick}\ 1; \mathsf{send}\ d_1\ \langle x_1 + 1, y_1\rangle; \\
& \mathsf{case}\ d_1\ \{\mathsf{ack} \hookrightarrow \ldots; x_2 \leftarrow \mathsf{recv}\ d_2; \mathsf{tick}\ 1; \mathsf{send}\ d_2\ \langle x_2 + 1, y_2\rangle; \\
& \qquad \mathsf{case}\ d_2\ \{\mathsf{ack} \hookrightarrow \ldots, \mathsf{timeout} \hookrightarrow \ldots\}, \\
& \mathsf{timeout} \hookrightarrow \ldots\}.
\end{aligned} \tag{D.2}$$

Section 6.2 studies a web server capable of scheduling sessions. With a round-robin scheduler, the server is

$$d_1 \leftarrow \mathsf{recv}\ c; d_2 \leftarrow \mathsf{recv}\ c; c \leftarrow P \leftarrow d_1, d_2,$$

where the process $c \leftarrow P \leftarrow d_1, d_2$ is implemented as

$$\begin{aligned} P := {} & x_1 \leftarrow \mathsf{recv}\ d_1; x_2 \leftarrow \mathsf{recv}\ d_2; \\ & \mathsf{tick}\ 1; \mathsf{tick}\ 1; \mathsf{send}\ d_1\ \langle x_1 + 1, y_1 \rangle; \mathsf{send}\ d_2\ \langle x_2 + 1, y_2 \rangle \\ & \mathsf{case}\ d_1\ \{\mathsf{ack} \hookrightarrow \mathsf{case}\ d_2\ \{\ldots\}, \mathsf{timeout} \hookrightarrow \mathsf{case}\ d_2\ \{\ldots\}\}. \end{aligned}$$

With the round-robin scheduler, resource-annotated A is

$$A_{\text{anno}} := \mathsf{int} \wedge \triangleright^1(\mathsf{int} \times \mathsf{int}) \supset \oplus\{\mathsf{ack} : \mathsf{int} \wedge \&\{\mathsf{success} : \triangleright^1 \mathbf{1}, \mathsf{failure} : \mathbf{1}\}, \mathsf{timeout} : \mathbf{1}\}.$$

Hence, the overall cost bound according to this resource-annotated session type is 4 (i.e. 2 for each instance of A). This is identical to the cost bound from Section 6.1, and the worst-case input generation algorithm generates the same worst-case input.

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany