Logical Methods in Computer Science
Volume 20, Issue 4, 2024, pp. 26:1–26:43
https://lmcs.episciences.org/

Submitted Sep. 06, 2023
Published Dec. 24, 2024

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 (DoS) 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. As pioneered by WISE [BJS09], symbolic execution is a well-studied technique for worst-case input generation. In WISE [BJS09] and SPF-WCA [LKP17], they first symbolically execute a program on all inputs of small sizes to identify a worst-case execution path p_{short}. This path is then generalized to a longer worst-case execution path p_{long} for a larger input. During the symbolic program along the execution path p_{long}, its corresponding path constraint is collected. Finally, solving the path constraint yields an inferred worst-case input for a large input size. Since these techniques do not explore the entire search space of large inputs, they are scalable but unsound.

To achieve soundness in worst-case input generation, Wang and Hoffmann [WH19] propose type-guided worst-case input generation for functional programming. In their

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

LOGICAL METHODS IN COMPUTER SCIENCE

DOI:10.46298/LMCS-20(4:26)2024

algorithm, symbolic execution of a functional program under analysis is guided by a *resource-annotated type* τ_{ra}, which is automatically inferred by the type-based resource analysis technique Automatic Amortized Resource Analysis (AARA) [HJ03, HH10, HAH12]. The resource-annotated type τ_{ra} encodes a sound (but not necessarily tight) polynomial worst-case bound. To identify a worst-case execution path, Wang and Hoffmann's algorithm searches for an execution path where the cost bound encoded by the resource-annotated type τ_{ra} is tight. Solving the path constraint of this worst-case execution path, we obtain a valid worst-case input: it has the same cost as the cost bound captured by the resource-annotated type τ_{ra}, which is not only sound but also tight.

All existing techniques for worst-case input generation, however, cannot handle 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 joint setting has a practical value. For example, denial-of-service (DoS) attacks overwhelm the memory of servers, which are typically concurrent programs. Hence, the worst-case input generation for concurrent programs under non-monotone resource metrics will be able to 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 extending their algorithm from functional programming to message-passing concurrent programming, the first challenge is to adapt the notion of *skeletons*, which specify the shapes and sizes of worst-case inputs to be generated, to message-passing concurrent programming. Designing skeletons is complicated by the following unique characteristics of message-passing concurrent 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 the shapes of inputs, if there are multiple, 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 all inputs are provided before program 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 message-passing 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.

Non-monotone resource metrics, wherein resources can be consumed as well as freed up, have two types of costs: *net cost* (i.e., the net quantity of resources consumed) and *high-water-mark cost* (i.e., the peak net cost that has been reached). For example, suppose 3 units of resources (e.g., memory cells) are consumed at the beginning of computation, but later, 2 units are freed up at the end of the computation. In this case, the net cost is $1 = 3 - 2$, while the high-water mark cost is 3 because, at any point during the computation, the maximum net cost is 3. In message-passing concurrent programming, a worst-case input is defined as an input with the maximum high-water-mark cost.

The second challenge in worst-case input generation for message-passing concurrent programming is to identify a worst-case schedule of concurrent processes, which is crucial for identifying a worst-case input. While the net cost of a program is independent of the schedules of concurrent processes, 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. Consequently, haphazardly performing symbolic execution of a concurrent program does not necessarily reveal the correct high-water-mark cost of a given execution path. We must additionally identify a worst-case schedule of concurrent processes. However, it is non-trivial to learn such a schedule on the fly during symbolic 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 the dependency of high-water-mark costs on schedules, we leverage *resource-annotated session types* [DHP18], whose resource annotations capture (i) sound bounds on high-water-mark costs and (ii) how many resources can be reused and transferred between processes. Thanks to the availability of sound high-water-mark cost bounds in resource-annotated session types, like Wang and Hoffmann's algorithm, our worst-case input generation algorithm is sound. Furthermore, the information about resource transfer enables us to correctly track high-water-mark costs of execution paths during symbolic execution.

To summarize, in this article, we make the following contributions:

- 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 prove 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 the channels c_1 and c_2 can be bidirectional. The process P uses the channel c_1 as a client and provides the channel c_2 as a provider. Fig. 1 (a) depicts the process P. The environment that the process P interacts with is called *the external*

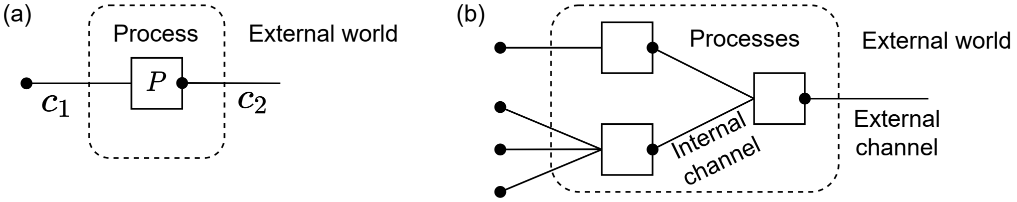

Figure 1. (a) Process P uses channel c_1 as a client and provides channel c_2. A dot on a channel denotes the channel's provider. (b) General SILL program consisting of multiple concurrent processes An internal channel connects two processes; an external channel connects a process with the external world.

world. Fig. 1 (b) depicts a general SILL program consisting of multiple concurrent processes. The network of concurrent processes in SILL must be tree-shaped: each channel is used by exactly one client, instead of being shared by multiple clients[1].

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

Suppose the process P is implemented as follows. The process 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 we need one memory cell for the key and another for the value. When the input stream on the channel c_1 terminates, all memory in the key-value store is released.

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

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

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

The session type of the channel c_2 is $\mathsf{int} \wedge \mathbf{1}$. It means the provider of the 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, resulting in resource-annotated session types. These resource annotations indicate the amount of *potential* necessary to pay for the computational cost, where the idea of potential comes from the potential method of amortized analysis for algorithms and data

[1] SILL's type system was designed to guarantee deadlock freedom. Because cyclic dependency among channels may cause a deadlock, SILL disallows channels from being shared. This restriction results in a tree-shaped network of concurrent processes. 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 that use the acquire-release primitives to achieve deadlock freedom while permitting the sharing of channels.

structures [Tar85]. Although potential and resources are similar to each other and hence can used interchangeably, they are subtly different: potential is an *abstract* resource used in the potential method, while resources typically refer to *concrete* computational resources such as time and memory. In our example, when a previously unseen IP address is encountered, two memory cells are allocated. Therefore, in the worst case, we need 2 units of potential to process each IP address on the channel c_1. This is why the resource-annotated session type of c_1 in Eq (2.1) contains $\triangleright^2$. It denotes that 2 units of potential are transferred from the channel client (i.e., the external world) to the channel provider (i.e., the process P).

Resource-annotated session types are inferred automatically. Because all numerical 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: the cost bounds represented by the inferred resource-annotated session types are guaranteed to be valid upper bounds on high-water-mark costs. Resource-annotated session types in resource-aware SILL can only encode linear cost bounds, but not polynomial ones[2].

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 non-monotone resource metrics (e.g., memory), where resources can be freed up as well as consumed, subsumes the definition of worst-case inputs for monotone resource metrics (e.g., running time). In monotone resource metrics, the net cost monotonically increases (without ever decreasing). Hence, in monotone resource metrics, the high-water-mark cost (i.e., the maximum net cost ever reached) 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, whose concrete values are to be determined later. Skeletons specify the shape of worst-case inputs to be generated.

For the 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 $x_1, \ldots, x_{10} \in \mathbb{Z}$ are integer-typed variables. This skeleton specifies that the external world should send ten cons's, followed by the label nil. As the channel c_2 does not take in any input from the external world, the channel c_2 does not need a skeleton.

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

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

[2] While resource-aware SILL [DHP18] only supports linear bounds, Automatic Amortized Resource Analysis (AARA) [HJ03, HH10, HAH12], which is an analogous type-based resource analysis method for functional programs, can infer multivariate polynomial bounds. Furthermore, all numerical constraints generated during the type inference in AARA are linear, even though it can infer multivariate polynomial cost bounds.

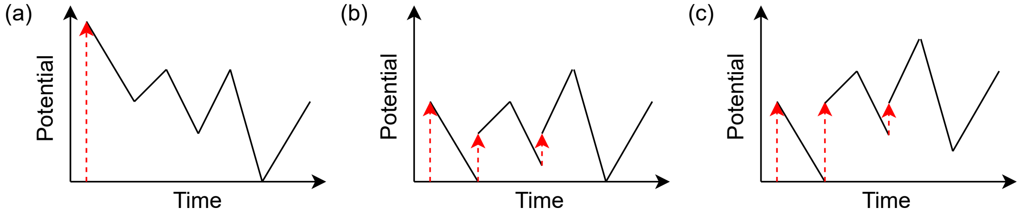

Figure 2. (a) Tight cost bound when all potential is supplied at once. The red dashed arrow indicates the potential supplied by the external world. (b) Tight cost bound when potential is supplied gradually. (c) Loose cost bound. Potential zero is never reached after the last injection of potential. So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

Symbolic execution. The goal of worst-case input generation is find an input whose high-water-mark cost achieves the cost bound inferred by resource-aware SILL. As resource-aware SILL guarantees the soundness of inferred cost bounds, once we find an input that has the same cost as an inferred cost bound, the input immediately qualifies as a worst-case input.

To find a worst-case input, we symbolically execute a program on a skeleton, searching 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 identifies a worst-case input. Suppose, for simplicity, that all necessary potential is supplied at the start of execution (Fig. 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), implying that the cost bound is not tight.

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

High-water-mark costs under concurrency. In the presence of multiple concurrent 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 types make the 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. The process P_1 has the high-water mark $h = 4$ and net cost $w = 0$. Also, the process P_2 has $(h, w) = (1, 0)$, i.e., it has the high-water mark $h = 1$ and the next cost $w = 0$. Next, the process P_1 sends the message to the process P_2, thereby synchronizing them. We assume that the communication between the processes P_1 and P_2 is *asynchronous*. That is, once it sends a message, the process P_1 does not need to wait for the process P_2 to receive the message. After sending the

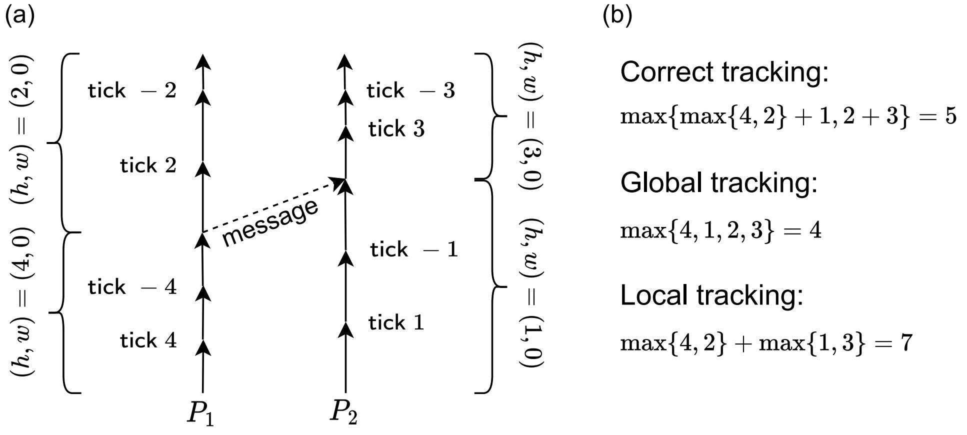

FIGURE 3. (a) Concurrent processes P_1 and P_2. Time passes by in the direction of arrows. (b) Results of global tracking and local tracking.

message, the process P_1 incurs $(h,w) = (2,0)$. After receiving the message, the process P_2 incurs $(h,w) = (3,0)$. This situation is depicted in Fig. 3 (a). In the figure, the annotation 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 the annotation tick q means $|q|$ units of resources are freed up. The arrows indicate the happened-before relation [Lam78].

The worst-case combined high-water-mark cost of the processes P_1 and P_2 is 5, and it can be derived as follows. Firstly, before the process P_2 receives the message, their worst-case combined high-water mark is $\max\{4,2\} + 1 = 5$. Due to the asynchrony of communication, by the time the process 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 $h = 1$. If the peak net costs of the processes P_1 and P_2 happen at the same time, their worst-case combined high-water mark is $\max\{4,2\} + 1 = 5$. Secondly, after the process P_2 receives the message, the combined high-water mark is $2 + 3 = 5$. This is because, in the worst case, the high-water marks of the processes P_1 and P_2 in their second phases, namely 2 and 3, happen at the same, resulting in the combined high-water mark of $2 + 3 = 5$. Therefore, overall, the worst-case combined high-water mark throughout the execution is $\max\{5,5\} = 5$.

Global and local tracking of high-water marks. To identify a worst-case execution path, we must calculate the tight worst-case high-water mark of any execution path. Two naive ways to track costs are 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 worst-case high-water mark (Fig. 3 (b)). On the one hand, global tracking may underestimate it. Before synchronization, if the peak costs of the processes 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 worst-case combined high-water mark. When the program terminates, the process P_1's

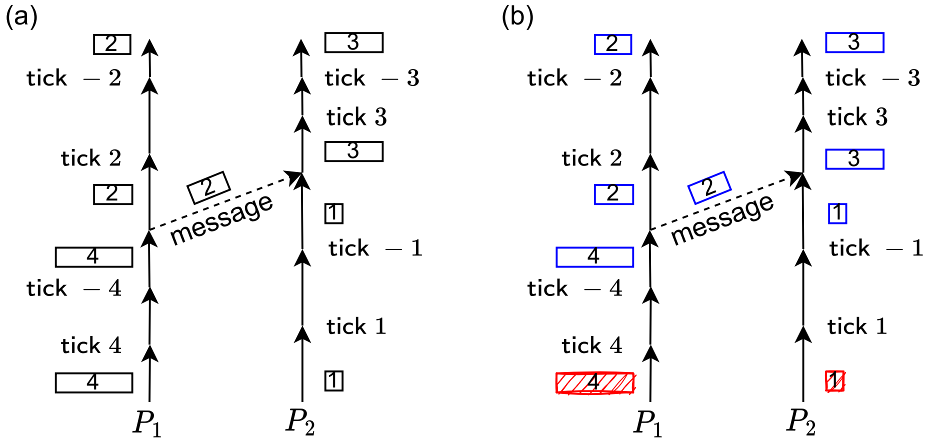

Figure 4. (a) The potential helps us derive the correct high-water mark. A rectangle next to an arrow represents the potential during the arrow's time period. The number inside a rectangle and its length indicate the amount of potential. If no rectangle exists, it indicates zero potential. (b) Same diagram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

local counter registers $\max\{4, 2\} = 4$, and P_2's counter registers $\max\{1, 3\} = 3$. Their sum is 7, which overestimates the tight worst-case combined high-water mark of 5.

The root cause of overestimation in local tracking is that it does not account for the possibility that, due to message-passing synchronization, the high-water marks of two concurrent processes cannot happen at the same time. For example, in Fig. 3 (a), the process P_1's local cost counter registers $(h, w) = (4, 0)$ *before* P_1 sends the message (and hence also *before* the process P_2 receives the message). Meanwhile, the process P_2's local cost counter registers $(h, w) = (3, 0)$ *after* the process P_2 receives the message. Thus, the high-water marks 4 (of the process P_1) and 3 (of the process P_2) cannot happen at the same time—these two high-water marks must respectively happen before and after the synchronization of the two processes. Nonetheless, because local tracking does not account for this fact, it naively sums the respective high-water marks of the two local counters, resulting in an overestimate of $3 + 4 = 7$ for the worst-case combined high-water mark.

Transfer of potential. To resolve this issue of local tracking, our key insight is to *transfer potential between processes*. Suppose the processes P_1 and P_2 initially store 4 units and 1 unit of potential, respectively. In the process P_1, after it runs tick 4, all potential stored in the process P_1 is consumed. But the expression tick -4 in the process P_1 frees up 4 units of potential. Likewise, the process P_2's potential is used up by the expression tick 1, but is later freed up by the expression tick -1. When the process P_1 sends the message, the message also carries 2 units of potential. This potential is paid by the process P_1, and in turn, it is used to pay for the process P_2's cost. Fig. 4 (a) depicts this situation.

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

What makes the potential method more powerful than local tracking is the ability to transfer potential between processes, allowing the potential to be reused (and hence shared) by multiple processes. By reusing and sharing the potential between processes, the local cost counters of concurrent processes are calibrated such that their sum yields a tighter combined high-water mark. To illustrate it, recall that, in Fig. 3, local tracking overestimates the worst-case high-water mark because local tracking does not account for the possibility that the high-water marks of two concurrent processes cannot happen at the same time. To fix this issue, we have the process P_1 send 2 units of potential to the process P_2, along the message, such that this potential can be reused by the process P_2. After the process P_2 receives the message, which carries the 2 units of potential, we use this potential to partially pay for the high-water mark of the process P_2 after the synchronization. This partial payment allows us to decrease the post-synchronization high-water mark of the process P_2 from 3 to $1 = 3 - 2$.

The key innovation of our worst-case input generation algorithm for concurrent programs is to leverage *resource-annotated session types* [DHP18], which capture information about potential transfer, to guide symbolic execution. We first automatically infer the resource-annotated session A of a given concurrent program expressed in SILL. We then run symbolic execution of the program to systematically and exhaustively search for a finite execution path (according to user-specified session skeleton like Eq (2.2)) where the bound on the high-water mark cost encoded by the resource-annotated session type A is tight. In addition to the high-water-mark bound, the type A encodes information about how resources are transferred between processes. Hence, by exploiting this information during the symbolic execution, for any execution path, we can tightly track the worst-case high-water mark (among all possible schedules of concurrent processes), thereby checking the tightness of the high-water mark bound encoded inside the type A.

Check tightness of cost bounds. Resource-annotated session types already capture sound bounds on high-water marks. In order for these bounds to translate into true high-water marks, it remains to ascertain that the bounds are tight. This is achieved 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 $\mathsf{tick}\ q$ 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 $\mathsf{tick}\ q$ for $q < 0$, then $|q|$ units of (blue) potential are freed up and become available to the process. A cost bound of an entire concurrent program is given by the total red potential supplied by the external world to the program.

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 Fig. 3, Fig. 5 (a) illustrates a situation where red potential is not consumed entirely. The processes 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 the process P_2, suggesting that the cost bound of 6 is not tight.

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 Fig. 5 (b). As in part (a), the processes P_1 and P_2 are initially given 4 units and 2 units of red potential, respectively. However, this time, the process P_1 only sends 1 unit, instead of 2 units, of (blue) potential to the process P_2. Consequently, 1 unit of the blue potential generated by $\mathsf{tick}\ {-4}$ on the process 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

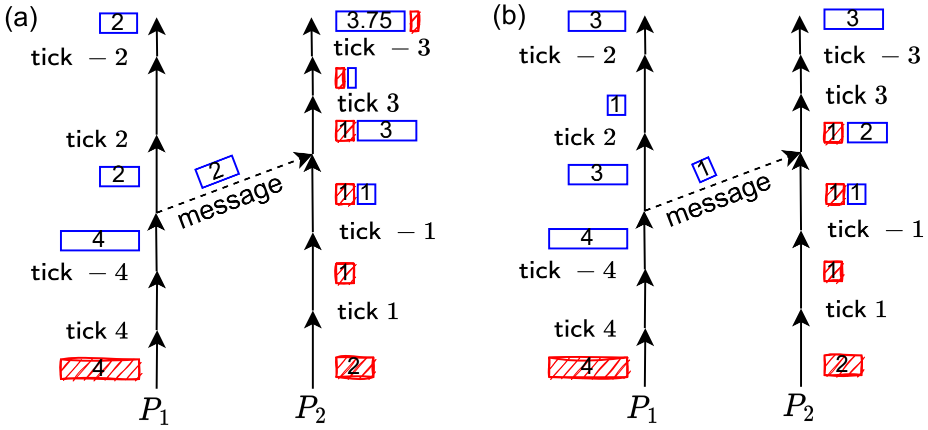

Figure 5. (a) Red potential in the process P_2 is not entirely consumed. Red hatched rectangles and blue blank rectangles represent red and blue potential, respectively. (b) Blue potential is generated in the process P_1 before red potential is consumed in the process P_2. But some blue potential is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

the process P_2. Hence, we could send more blue potential from the process P_1 to the process P_2, thereby substituting the blue potential for red potential supplied to the process P_2 by the external world. This means the cost bound of 6 is, again, not tight.

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

Solve 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 Eq (2.1), the worst-case execution path is where each incoming IP address requires 2 units of potential. Solving the path constraint of this path, we obtain a set of mutually distinct IP addresses such as Eq (2.3).

Current SMT technologies cannot solve constraints over an infinite data structure. Consequently, it is necessary to require the input portion of a session skeleton to be finite; otherwise, we would not be able to find an infinitely large worst-case input by solving its path constraint using an SMT solver. An example of a path constraint over an infinite worst-case input that modern SMT solvers cannot handle is given in Eq (4.3) (Section 4.3).

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: a provider and a 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.

[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].

Channels are typed with (resource-annotated) session types, which describe the communication protocols on the channels. Well-typedness of channels in SILL 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 τ in the functional layer and resource-annotated session types A in the process layer are formed by the following grammar:

$$\begin{array}{rll}
b & ::= \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid b_1 \times b_2 \mid b_1 + b_2 & \text{primitive and polynomial types} \\
\tau & ::= \{c : A \leftarrow \overline{c_i : A_i} \mathbin{@} q\} & \text{process type} \\
 & \quad \mid b \mid \tau_1 \to \tau_2 & \text{functional types} \\
A & ::= X \mid b \supset A \mid \cdots \mid \triangleright^q A & \text{resource-annotated session types (Table 1).}
\end{array}$$

Functional types. In the functional layer, if an expression e has a process type $\{c : A \leftarrow \overline{c_i : A_i} \mathbin{@} q\}$, the expression 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 $A_1, \ldots, A_n$. The annotation $q \in \mathbb{Q}_{\geq 0}$ denotes the constant potential initially stored in the process. The rest of the types in the functional layer are standard.

Session types. In Table 1, the first column lists all resource-annotated session types. The second column lists the continuations of the session types from the first column, that is, what session type the channel will have after it makes one action (e.g., send or receive a message). The last column describes, for each session type A, the operational semantics of a channel c with the session type A from the viewpoint of the the channel c's provider.

Briefly, the session type $b \supset A$ (and $b \wedge A$) receives (and sends) a value of a functional type b and then proceeds to the session type A. The session type $A_1 \multimap A_2$ (and $A_1 \otimes A_2$) receives (and sends) a channel of session type A_1 and then proceeds to the session type A_2. The session type $\&\{\overline{\ell_i : A_i}\}$ (and $\oplus\{\overline{\ell_i : A_i}\}$) receives (and sends) a label ℓ_j for some j and proceeds to the session type A_j accordingly. The session type $\mathbf{1}$ closes the channel and sends the message end to the channel client in order to signal the closure. The session type $\triangleleft^q A$ (and $\triangleright^q A$) receives (and sends) $q \in \mathbb{Q}_{>0}$ units of potential to the channel client.

Unlike the original SILL [TCP13, PG15], resource-annotated SILL [DHP18] does not offer the exponential operator ! for copying a channel. Hence, resource-aware SILL is related, via the Curry-Howard correspondence, to intuitionistic multiplicative-additive linear logic (IMALL) [LSS93].

We assume a global signature Σ containing definitions of type variables of 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], that is, we cannot have a recursive session type $\mu X.X$ that remains identical after unfolding the recursive operator. 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 attempts to automatically determine resource annotations $q \in \mathbb{Q}_{>0}$ in $\triangleright^q$ and $\triangleleft^q$ [DBHP19] by collecting linear constraints from according to the type system in Section 3.4 and solving them by a linear-program (LP) solver. So a user does not need to manually provide the values of q.

Table 1. Resource-annotated session types and process terms. In the last column, their operational semantics are described from the viewpoint of channel providers. For the session type $\mathbf{1}$, the first row is for the provider of the channel, and the second row is for the client of the channel.

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

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

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

Functional terms. The term $c \leftarrow P_{c,\overline{c_i}} \leftarrow \overline{c_i}$ encapsulates a process $P_{c,\overline{c_i}}$ that provides a channel c and uses channels $\overline{c_i}$. The rest of the functional layer's syntax is standard.

Processes. In the process layer, the process $c \leftarrow e \leftarrow \overline{c_i}; P_c$ first spawns a new process denoted by e. This child process provides a channel c and uses channels $\overline{c_i}$ as a client. After spawning e, the parent process continues as a process P_c and uses the channel c as a client. The process $c_1 \leftarrow c_2$ forwards messages between channels c_1 and c_2 in both directions. The process $\mathsf{tick}\ q; P$ consumes $q \in \mathbb{Q}$ units of resources. The construct $\mathsf{tick}\ q$ is inserted either manually by a user or automatically according to a resource metric of the user's interest. 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 the third and fourth columns of Table 1. The value of q in processes $\mathsf{get}\ c\ \{q\}; P$ and $\mathsf{pay}\ c\ \{q\}; P$ is automatically inferred [DHP18].

For illustration, consider a process P that provides a channel c and uses no channels. Suppose that the process P receives an integer $x \in \mathbb{Z}$ on the channel c and spawns a new process that (i) provides a new channel f and (ii) sends an integer $x + 1$. Lastly, the process

P closes the channel c. An implementation of the process P is

$$P \coloneqq x \leftarrow \mathsf{recv}\ c; d \leftarrow f\ x; \mathsf{close}\ c \tag{3.1}$$

$$f(x) = \mathsf{send}\ c\ (1 + x); \mathsf{close}\ c. \tag{3.2}$$

The definition (3.1) of the process P calls a function $f : \mathsf{int} \to \{c \leftarrow \cdot\ @\ 0\}$, whose type signature means that the function f takes an integer as input and returns a process of type $\{c \leftarrow \cdot\ @\ 0\}$, i.e., it provides a channel c, uses no channels, and requires zero initial potential. When the process P runs $d \leftarrow f\ x$, the channel d is substituted for the channel c used inside the function body (3.2) of f.

A resource metric is said to be monotone if every $\mathsf{tick}\ q$ has $q \geq 0$. Conversely, if $q < 0$ (i.e., resources are freed up as well as consumed) 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

$$\frac{I_1 \quad I_2 \quad \cdots \quad I_n}{J_1 \quad J_2 \quad \cdots \quad J_m} \qquad (n, m \in \mathbb{N}). \tag{3.3}$$

If the predicates $I_1, \ldots, I_n$ *all* 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 $\mathsf{msg}(c, w, M)$. The predicate $\mathsf{proc}(c, w, P)$ represents a process P providing a channel c. The predicate $\mathsf{msg}(c, w, M)$ means a message M is being transferred across a channel c, but has not been received yet. In the predicate $\mathsf{proc}(c, w, P)$, the net-cost counter $w \in \mathbb{Q}$ tracks the cumulative net cost of the process. In the predicate $\mathsf{msg}(c, w, M)$, the number $w \in \mathbb{Q}$ is used when a process terminates and transfers the net cost at that point to another process.

Fig. 6 displays some key rules of the substructural cost semantics. The grammar of a message M and the remaining rules of the cost semantics can be found in Fig. 12 (Appendix A.1).

The rule spawn considers a configuration where (i) we have a process $c \leftarrow e \leftarrow \overline{c_i}; Q_c$ (in the first premise) and (ii) the functional term e evaluates to a value $x \leftarrow P_{x,\overline{x_i}} \leftarrow \overline{x_i}$ (in the second premise), where $P_{x,\overline{x_i}}$ is a process. The two conclusions in the rule spawn indicate that the next configuration replaces the premises with the following predicates: (i) $\mathsf{proc}(c', 0, P_{c',\overline{c_i}})$ for the newly spawned process where the net-cost counter w is initialized to zero and (ii) $\mathsf{proc}(d, w, Q_{c'})$ for the parent process.

The rule $\supset S$ concerns a configuration containing a process $\mathsf{send}\ c\ e; P$ (in the first premise), which sends a message e on channel c. To send a message in the operational semantics, the premises in the rule are replaced with these two predicates: (i) $\mathsf{proc}(d, w, P[c'/c])$ for the continuation of the process after sending a message and (ii) $\mathsf{msg}(c', 0, \mathsf{send}\ c\ v; c' \leftarrow c)$ for the message $\mathsf{send}\ c\ v; c' \leftarrow c$ containing the content v that is being transferred across a freshly renamed channel c'. The message predicate $\mathsf{msg}(c', 0, \mathsf{send}\ c\ v; c' \leftarrow c)$ is then handled by the rule $\supset R$, which concerns a process ready to receive a message (in the first

$$\boxed{\mathsf{proc}(c, w, P) \text{ and } \mathsf{msg}(c, w, M)}$$

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

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

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

$$\dfrac{\mathsf{proc}(c, w, \mathsf{close}\ c)}{\mathsf{msg}(c, w, \mathsf{close}\ c)}\mathbf{1}S \qquad \dfrac{\mathsf{msg}(c, w_1, \mathsf{close}\ c) \qquad \mathsf{proc}(d, w_2, \mathsf{wait}\ c; P)}{\mathsf{proc}(d, w_1 + w_2, P)}\mathbf{1}R$$

Figure 6. Key rules of the substructural cost semantics of resource-aware SILL. The functional-layer evaluation judgment $e \Downarrow v$ in the rules spawn and $\supset S$ means the functional term e evaluates to a value v. The definition of the functional-layer evaluation judgment $e \Downarrow v$ is standard. The remaining rules are available in Fig. 12 (Appendix A.1).

premise). 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 rule tick states that, whenever tick q is executed, the net-cost counter w in the premise $\mathsf{proc}(c, w, \mathsf{tick}\ q; P)$ is incremented by q. The rule $\mathbf{1}S$ concerns a process close c, which seeks to close the channel c that the process provides (and hence also terminates the process itself). To close the channel in the operational semantics, we remove replace this premise with a predicate $\mathsf{msg}(c, w, \mathsf{close}\ c)$ that carries (i) a signal for the channel closure to the channel client and (ii) the final net cost w of the process. The client then receives the message, together with the final net cost of the sender, according to the rule $\mathbf{1}R$.

The net cost of a configuration C is the sum of the net-cost counters w in predicates $\mathsf{proc}(c, w, P)$ and $\mathsf{msg}(c, w, M)$ in the configuration C when the SILL program terminates (i.e., no more rewrite rules can be applied to the configuration C). The high-water mark of the configuration C is its maximum net cost that has ever been reached (i.e., the maximum sum of the net-cost counters w in all predicates in the configuration C) during the execution of the SILL program.

3.4. **Type System.**

Fix a signature Σ containing type definitions and function definitions. A typing judgment of the process layer has the form

$$\Phi; \Delta; q \vdash P :: (c : A). \tag{3.4}$$

The judgment (3.4) means the channel c is provided by the process P and has a resource-annotated session type 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 the process P as a client. The number $q \in \mathbb{Q}_{\geq 0}$ denotes how much potential is initially stored in the process P.

Fig. 7 displays key rules of resource-aware SILL's type system. The remaining rules are in Fig. 13 (Appendix A.1).

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

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

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

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

FIGURE 7. Key rules of the type system of resource-aware SILL. The judgment $\Phi \vdash e : \tau$ in the rules spawn and $\supset L$ is a functional-layer typing judgment stating that the functional term e has a functional type τ. The remaining rules are in Fig. 13 (Appendix A.1).

The rule spawn in Fig. 7 states that a process $c \leftarrow e \leftarrow \overline{c_i}; Q_c$, which spawns a new process-term e, is well-typed if e has a correct process type (in the first premise) and the continuation process Q_c is well-typed (in the third premise). Additionally, the initial potential necessary for the process $c \leftarrow e \leftarrow \overline{c_i}; Q_c$ is $p + q$, where p is the initial potential for the spawned process e and q is the initial potential for the continuation process Q_c.

The rule $\supset L$ concerns a process send $c\ e; P$, which sends a value e of the functional type b on the channel c used by the process as a client. The dual rule, $\supset R$, concerns a process $x \leftarrow$ recv $c; P_x$, which is ready to receive a message of the functional type b on the channel c provided by the process. The rule relax weakens the resource annotation q in the judgment: if the initial potential q is sufficient for the process P to run, then any larger potential $q > p$ works as well. The rule tick states that, to run the construct tick $q; P$, we require the initial potential of $p + q$, where p is the potential necessary for the continuation process P.

Thm. 3.1 states the soundness of the type system of resource-aware SILL: given an initial configuration C where the net cost is zero, the total potential of the configuration C is an upper bound of the high-water mark cost for any configuration reachable from C.

Theorem 3.1 Soundness of resource-aware SILL [DHP18, DBHP19]**.** *Given a well-typed configuration initial C whose net cost is zero, let $p \in \mathbb{Q}$ be the total potential stored in the configuration C. The total potential in the configuration C is defined as the sum of all q in judgments $\Phi; \Delta; q \vdash P :: (c : A)$ for all processes P in the predicates* $\mathsf{proc}(c, \_, P) \in C$. *If $C \to^* C'$ (i.e., a configuration C' is reachable from the configuration C) and $h \in \mathbb{Q}$ is the high-water-mark cost of the configuration C', then $h \leq p$ holds. That is, the initial potential p correctly upper-bounds the high-water mark cost h.*

To prove Thm. 3.1, Das et al. [DHP18] first define a typing judgment $\Phi; \Delta_1 \overset{E}{\models} C :: \Delta_2$ for a configuration C, where Φ is a functional-layer typing context and Δ_i $(i = 1, 2)$ is a process-layer typing context. The judgment defines $E \in \mathbb{Q}$, called energy, as the sum of the total potential and the net cost in the configuration C. Das et al. then prove that, as the configuration C evolves, the energy E is conserved: it remains the same or decreases (due to the weakening of potential). This is the resource-aware SILL's equivalent of the classic type preservation theorem, and it implies Thm. 3.1. 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.

4. Session Skeletons

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

Message-passing concurrent programming poses three challenges in the design of skeletons. First, in the presence of multiple channels, their skeletons are interdependent on each other due to the interaction between channels. Second, inputs to concurrent programs may be infinite. Third, the input and output are intertwined in such a way that the 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 this article because, to fully address this challenge, it is necessary to enrich session types such that they capture more information, particularly the interdependence between input and output. The challenge does not affect the relative-completeness theorem of our worst-case input generation algorithm (Thm. 5.9), since the theorem assumes the cost bounds, which we calculate by simply summing all resource annotations in session skeletons, are tight.

4.1. Syntax.

Fix a set $\mathcal{X}_{\text{skeleton}}$ of skeleton variables. They are placeholders for concrete values in a worst-case input. Skeletons are formed by the following grammar:

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

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

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

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

$$\overset{\text{K:ChannelIn}}{\frac{\Phi \vdash A_1 \leqslant K_1 \qquad \Phi \vdash K_2 \leqslant A_2}{\Phi \vdash K_1 \multimap K_2 \leqslant A_1 \multimap A_2}} \qquad \overset{\text{K:ChannelOut}}{\frac{\Phi; \Delta \vdash K_1 \leqslant A_1 \qquad \Phi; \Delta \vdash K_2 \leqslant A_2}{\Phi; \Delta \vdash K_1 \otimes K_2 \leqslant A_1 \otimes A_2}}$$

$$\frac{\forall j \in N'. \Phi \vdash K_j \leqslant A_j \qquad \emptyset \subset N' \subseteq N}{\Phi \vdash \&_x\{\ell_i : K_i \mid i \in N'\} \leqslant \&\{\ell_j : A_j \mid j \in N\}}\text{K:ExtChoice} \qquad \frac{\Phi; \Delta \vdash K \leqslant A}{\Phi; \Delta \vdash \triangleleft^q K \leqslant \triangleleft^q A}\text{K:Get}$$

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

Figure 8. Inference rules of the compatibility relation between a session skeleton and a resource-annotated session type. The judgment $\Phi \vdash H : b$ in K:ValIn means the functional-layer skeleton H has a functional type b.

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 the skeleton K with the resource-annotated session type 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 the skeleton K is a valid skeleton of the session type A, given that the external channel c is provided by a process. Fig. 8 defines Eq (4.1). Dually, 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 Fig. 8.

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 the skeleton K admits some of the semantic objects of the session type A, the skeleton K can be considered as a subtype of the session type A.

Due to the rule K:Ter, given a session skeleton $K = \mathbf{1}$, the session type $\mathbf{1}$ is the only compatible session type. Hence, a session skeleton K is disallowed from stopping halfway when the corresponding session type A has not terminated yet, e.g., $\Phi \not\vdash \mathbf{1} \leqslant \mathsf{int} \wedge \mathbf{1}$.

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

In the rule K:ChannelIn, the first premise uses the dual judgment. This is because, in the session type $A_1 \multimap A_2$, the input session type A_1 reverses the roles of the channel provider and client. In the rule K:ExtChoice, a 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 a session type $\&\{\ell_i : A_i \mid i \in N\}$ is made by the external world. Therefore, the skeleton is allowed to limit the set of i to choose from.

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 non-terminating process $\cdot; 0 \vdash P :: (c : A)$, where[4]

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

The process P is willing to accept two labels. If first is chosen, every iteration on channel c requires 2 units of potential. Otherwise, if second is chosen, every iteration only needs 1 unit of potential.

It is unclear which scenario should be deemed the worst-case input. One possible answer is that they both have an equal cost of infinity. 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. However, this idea sounds too simplistic. Another possible answer is that the first branch results in a higher cost than second because the former entails 2 units of cost per iteration, while the latter incurs only 1 unit of cost per iteration. Therefore, at any moment in time, the first branch has a higher cumulative cost than the second branch. However, this reasoning implicitly treats each iteration in Eq (4.2) equally. But it is arguable whether the iterations inside the two branches in Eq (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.

Generating infinite data structures. It is tricky to solve constraint satisfaction problems for infinite data structures. By way of example, consider a process $\cdot; 0 \vdash P :: (c : A)$, where $A \coloneqq \mu X.\mathsf{bool} \multimap X$. Suppose the process P is implemented such that the only worst-case input is an alternating sequence of true and false. To encode an infinite stream of Booleans, a sensible idea is to use 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. The path constraint for the worst-case input, where true and false alternate, is

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

Here, $f : \mathbb{N} \to \mathsf{bool}$ is an uninterpreted function, and we seek a concrete f that satisfies Eq (4.3). Unfortunately, the current SMT technologies are incapable of finding suitable f in this example. We tested SMT solvers Z3 [dMB08] and CVC4 [BCD+11] on the SMT-LIB2 encoding of Eq (4.3) (Appendix B.1), and neither of them could verify the satisfiability.

[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 $X = \mu X.A_X$.

To go around these two challenges of infinite worst-case inputs, we require the input portion of a skeleton to be finite. Appendix B.1 provides further details.

4.4. **Input Generation with Loose Cost Bounds.** In the presence of multiple channels, it is non-trivial to calculate a precise cost bound due to the intertwining of the input and output across different channels. For illustration, consider a process P with two external channels

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

where

$$A_1 \coloneqq \oplus\{\mathsf{expensive} : \triangleright^{2} \mathbf{1}, \mathsf{cheap} : \mathbf{1}\} \qquad A_2 \coloneqq \oplus\{\mathsf{expensive} : \triangleleft^{3} \mathbf{1}, \mathsf{cheap} : \mathbf{1}\}. \tag{4.5}$$

The external world first chooses between $\mathsf{expensive}$, which requires 2 units of potential, and cheap, which requires no potential. The process P next chooses between $\mathsf{expensive}$ and cheap. If the process P chooses $\mathsf{expensive}$, 3 units of potential are sent as input to the process P; otherwise, no extra potential is required.

The input and output are intertwined in this example. The process 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 the channel c_1, while the output and second input (i.e., 3 units of potential) happen on the channel c_2.

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

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

$$P \coloneqq \mathsf{case}\ c_1\ \{\mathsf{expensive} \hookrightarrow \mathsf{tick}\ 2; c_2.\mathsf{cheap}, \mathsf{cheap} \hookrightarrow \mathsf{tick}\ 3; c_2.\mathsf{expensive}\}. \tag{4.6}$$

The tight cost bound is 3, which is lower than the bound deduced from Eq (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 (4.6). This is evidenced by another valid typing judgment of the process P:

$$c_1 : A; 3 \vdash P :: (c_2 : A), \tag{4.7}$$

where $A \coloneqq \oplus\{\mathsf{expensive} : \mathbf{1}, \mathsf{cheap} : \mathbf{1}\}$. In (4.7), all necessary potential comes from the initial constant potential 3 stored in the process P, which is tighter than the cost bound of 5. Thus, typing judgments can misrepresent cost bounds, as exemplified by Eq (4.4), even when resource-aware SILL is capable of deriving tight cost bounds.

The root cause is that session types are not rich enough to capture information about the interdependence between input and output. Suppose resource annotations are scattered over a session type. When inputs, including the supply of incoming potential, are interspersed with outputs, an early input affects an output, which in turn affects a later input's resource-annotated session type. Consequently, some combinations of inputs may be infeasible.

Furthermore, if the 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. In the above example, if the type system returns the typing judgment (4.7), we obtain a precise cost bound of 3. However, if the type system returns the typing judgment (4.4), which is an equally valid typing judgment, we obtain a loose cost bound of 5. Thus, 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. Therefore, to calculate a precise cost bound, it is not sufficient to just examine the resource annotations in session types and session skeletons.

Addressing this issue is beyond the scope of this article. For simplicity, when facing a choice between branches, we sum resource annotations in session skeletons to obtain the branches' respective cost bounds and pick the branch with a higher cost bound. The example (4.4) still respects relative completeness of worst-case input generation (Thm. 5.9) because the relative-completeness 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 Alg. 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 the skeletons $K$ satisfy all requirements
4:     Run symbolic execution while tracking potential to identify an execution path where
       the cost bound is tight. Also, construct a path constraint $\phi$ of this execution path
5:     Solve the path constraint $\phi$ from the previous step . If the path constraint is infeasible,
       go back to line 4 to search for another execution path where the cost bound is tight.
```

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

- The session skeletons K are compatible with their original session types (Section 4.2).
- The input portions of the session skeletons K are finite (Section 4.3).

Section 5.1 describes how we track potential during the symbolic execution (line 4). Section 5.2 formalizes the symbolic execution. If the symbolic execution finds an execution path where the cost bound is tight, the corresponding path constraint ϕ is fed to an SMT solver in line 5. If the path constraint ϕ is solvable, we obtain a concrete worst-case input. Otherwise, if the path constraint ϕ is infeasible (i.e., it has no solutions), we go back to line 4, searching for another execution path where the cost bound is tight.

5.1. Checking the Tightness of Cost Bounds.

The symbolic execution searches for an execution path where the cost bound is tight. Potential in SILL has a local nature: the 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 $q < 0$).

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

- 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 a Boolean flag r_1) to another process (which initially has a flag r_2). The flag of the sender becomes $r_1 \vee (p = 0)$, and the flag of the recipient becomes $r_1 \wedge r_2$.

We forbid processes from throwing away potential when $r = \mathsf{false}$. Potential is thrown away by the rule relax in the type system (Fig. 7). If it happens, it means the cost bound is not tight. In such an event, the 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.

Blue potential. Blue potential must be consumed if its generation precedes the consumption of red potential. To formally define what it means for an event (e.g., sending and receiving of messages, and generation and consumption of potential) to precede another, we introduce the happened-before relation → between events [Lam78], which is a well-established notion in distributed and concurrent computing.

Pictorially, the happened-before relation is illustrated in Fig. 3. The figure contains arrows within each of the processes P_1 and P_2 and another arrow between them (for sending and receiving a message). These arrows indicate the chronological ordering of events (regardless of how concurrent processes are scheduled). Taking the transitive closure of these arrows yields the happened-before relation →.

Definition 5.1 (Happened-before relation [Lam78])**.** The happened-before relation → is the smallest binary relation between events (e.g., sending and receiving messages) that is closed under the following three conditions. First, if an event A happens before an event B on the same process, $A \to B$ holds, i.e., the event A precedes the event B. Second, if an event A sends a message and an event B receives the message, $A \to B$ holds. Third, if $A \to B$ and $B \to C$ are true, so is $A \to C$.

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 an expression $\mathsf{tick}\ q$, where $q < 0$, in a 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.

Defn. 5.2 defines what it means for one process to be synchronized with another process since some event.

Definition 5.2 (Synchronization of two processes)**.** A process Q is synchronized with another process P since some event of the process P if and only if the process Q's current state happens after the event.

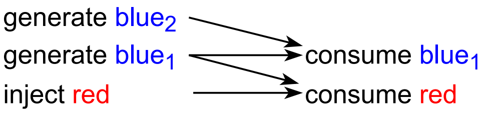

Figure 9. Alternating chain of the happened-before relation between events. The label "inject red" refers to an event of injecting red potential to some process from the external world. The label "generate blue" refers to the generation of blue potential.

For concreteness, suppose that a process P has generated blue potential of the ID blue$_1$. During the symbolic execution, we track the following items:

- The set of processes that have been synchronized (Defn. 5.2) with the process P since the generation of the ID blue$_1$. We track such processes by passing around the ID blue$_1$ whenever a message is sent by the process P or other processes that carry the ID 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 the process P.
- Whether the potential blue$_1$ is completely consumed.

To understand how tracking these items helps us detect loose cost bounds, consider a scenario where (i) a process Q has been synchronized with the process P; (ii) the process Q consumes red potential; and (iii) the potential blue$_1$ is not entirely consumed by the end of program execution. Because the potential blue$_1$ is generated before the potential red is consumed, we can substitute blue potential for red potential, thereby lowering the amount of necessary red potential without plunging into negative potential. Hence, the cost bound is not tight in this case.

More generally, we can substitute blue potential for red potential along a chain of distinct blue potential, each with a different ID. For instance, we want to substitute certain blue potential (with ID, say, blue$_1$) for red potential, where the potential blue$_1$ was generated before the consumption of red potential. However, the potential blue$_1$ is entirely consumed. So we must find another blue potential (say blue$_2$) that can substitute for the potential blue$_1$. This is possible when (i) the potential blue$_2$ is generated before the potential blue$_1$ is consumed and (ii) the potential blue$_2$ is not entirely consumed. Fig. 9 depicts this situation where we have two distinct potential, namely blue$_1$ and blue$_2$, whose generation and consumption are related by the happened-before relation.

If the potential blue$_2$ is not completely consumed, we can push the leftover potential of blue$_2$, from "generate blue$_2$" to "inject red," along the alternating path of $\rightarrow$ in Fig. 9. As a result, we can reduce the total red potential injected to the program without plunging into negative potential, thereby lowering the cost bound.

To correctly detect such alternating paths of the happened-before relation $\rightarrow$, 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 a node blue$_i$ to a node red such that the node blue$_i$ is not entirely consumed, then the cost bound is not tight. If the cost bound is detected to be loose on the current execution path, the symbolic execution backtracks and explores another execution path.

Thm. 5.3 states the correctness of tracking potential. A proof is given in Appendix C.2.

Theorem 5.3 (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.** The symbolic execution runs 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 the symbolic execution. The symbolic execution involves two types of predicates:

$$\mathsf{proc}(\Delta; q \vdash P :: (c : A), \phi, \mathrm{IDs}) \qquad \mathsf{msg}(c, M, \phi, \mathrm{IDs}). \tag{5.1}$$

The first predicate of Eq (5.1) 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). A logical formula ϕ is a path constraint so far and will later be fed to an SMT solver. The component $\mathrm{IDs} = (\mathrm{IDs}_s, \mathrm{IDs}_p)$ 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 the set IDs_s of a process P contains an ID blue_i, then the process P's current state happens after the potential blue_i was generated. The second set $\mathrm{IDs}_p \subset \mathcal{C}$ tracks potential transfer: if the set IDs_p of a process P contains an ID i, it means the process P contains the (red or blue) potential identified by the ID i.

The second predicate $\mathsf{msg}(c, M, \phi, \mathrm{IDs})$ in Eq (5.1) represents a message M (encoded as as process) that provides a channel c. A logical formula ϕ is a path constraint carried by the message.

Fig. 10 displays key rewriting rules. The remaining rules are given in Appendix C.4.

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

The rule $\& R_{\text{external}}$ resolves an external choice on an external channel. A label $k \in N$ is chosen such that the skeleton A_k has the highest cost bound among $\{A_i \mid i \in N\}$. The path constraint ϕ is then augmented with a constraint $x = k$, indicating that the external world should choose $k \in N$ to trigger the worst-case behavior. If we find out later that the current execution path's cost bound is not tight, we backtrack and try a different $k' \neq k$ such that $A_{k'}$ has the highest cost bound. If the algorithm fails to find any A_k $(k \in N)$ with the highest cost bound that is tight, then algorithm returns no worst-case inputs.

The rule $\mathbf{1}_S$ forbids red potential from being wasted: red potential must not remain when a process terminates. Likewise, the rule $\triangleleft L_{\text{external}}$, which is for an external channel, forbids red potential from flowing back to the external world. Furthermore, if we waste blue potential (i.e., $\mathrm{IDs}_p \setminus \{\mathrm{red}\} \neq \emptyset$) in the rules $\mathbf{1}_S$ 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 the rule $\triangleleft R_{\text{external}}$, the process receives red potential from the external world. So the ID red is added to the set IDs_p of the recipient.

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

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

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

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

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

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

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

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

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

5.3. Soundness and Relative Completeness.

Thm. 5.3 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.

Cost bounds encoded by resource annotations. Defn. 5.4 defines the cost bound of a resource-annotated skeleton.

Definition 5.4 (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 as follows.

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

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 Defn. 5.4 to make sense, the input portion of the skeleton K must be finite. $\triangleleft^q$ contributes to cost bounds in Defn. 5.4, but $\triangleright^q$ does not. If the cost bounds factored in outgoing potential (i.e., $\triangleright^q$) as well as incoming potential (i.e., $\triangleleft^q$), the 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.

Definition 5.5 (Cost bounds of configurations)**.** Suppose C is a configuration with external channels $c_1, \ldots, c_m, c_{m+1}, \ldots, c_n$. Here, the channels $c_1, \ldots, c_m$ are provided by processes in the configuration C, whereas the channels $c_{m+1}, \ldots, c_n$ 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 the configuration C. The cost bound of the configuration C is defined as

$$\lceil \bowtie \rceil(C) \coloneqq p + \sum_{i=1}^{m} \lceil \triangleright \rceil(K_i) + \sum_{i=m+1}^{n} \lceil \triangleleft \rceil(K_i). \tag{5.2}$$

Here, $\lceil \triangleright \rceil(K_i)$ denotes the cost bound of a skeleton K_i when the external channel is provided by some process in the network (Defn. 5.4). The dual cost bound is $\lceil \triangleleft \rceil(K_i)$.

Similarity relation and simulation. Defn. 5.6 defines a similarity relation between predicates. This relation can be lifted from predicates to configurations (Defn. C.5).

Definition 5.6 (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 the symbolic execution. The similarity relation $\sim$ between a predicate in symbolic execution and a predicate in the cost semantics is defined by

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

Here, the premise $S \vdash P_{\mathrm{sym}} = P_{\mathrm{cost}}$ means a predicate P_{sym} in the symbolic execution and a predicate P_{cost} in the cost semantics are identical under the mapping S.

Prop. 5.7 establishes a simulation between the symbolic execution and cost semantics. A proof is given in Appendix C.3.

Proposition 5.7 (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 the cost semantics. These configurations satisfy two conditions:* (i) *$C_{1,sym}$ transitions to $C_{2,sym}$ in one step of the symbolic execution and* (ii) *$C_{1,sym} \sim C_{1,cost}$ holds. Then there exists a configuration $C_{2,cost}$ of the cost semantics such that the following diagram commutes:*

$$\begin{tikzcd}
C_{1,sym} \arrow[r, "w"'] \arrow[d, squiggly, no head] & C_{2,sym} \arrow[d, squiggly, no head] \\
C_{1,cost} \arrow[r, "w"', "\leq 1"] & C_{2,cost}
\end{tikzcd} \tag{5.3}$$

In the commutative diagram (5.3)*,* $C_{1,sym} \underset{w}{\rightarrow} C_{2,sym}$ *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 (Defn. 5.5). Likewise,* $C_{1,cost} \underset{w}{\rightarrow} C_{2,cost}$ *means the configuration* $C_{1,cost}$ *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 Prop. 5.7, one transition step in symbolic execution may correspond to zero steps in the cost semantics. It happens when we transfer potential by rewriting rules such as $\lhd L_{\text{external}}$ in the symbolic execution, which do not have corresponding rules in the cost semantics.

Thm. 5.8 states the soundness of the worst-case input generation algorithm. In the statement of the theorem, $[\![K_1, \dots, K_n]\!]$ denotes the set of all possible inputs conforming to session skeletons $K_1, \dots, K_n$ on channels $c_1, \dots, c_n$. Each input is a multiset of predicates $\mathsf{msg}(c, M)$ for channels c and messages M. A formal definition of the set of possible inputs is given in Defn. C.1.

Theorem 5.8 (Soundness of worst-case input generation)**.** *Given a collection* $K_1, \dots, K_n$ *of skeletons, suppose the symbolic execution algorithm successfully terminates. Let* ϕ *be a path constraint generated by the symbolic execution and* $t \in [\![K_1, \dots, K_n]\!]$ *be an input satisfying* ϕ*. Then* t *has the highest high-water-mark cost of all inputs from* $[\![K_1, \dots, K_n]\!]$*.*

Proof. Prop. 5.7 shows that the symbolic execution correctly tracks of potential and net cost: both of them change by the same amount (but in the opposite direction). Furthermore, by Thm. 5.3, the high-water-mark cost bound is tight for the solution t to the path constraint ϕ. Therefore, the high-water-mark cost of the input t to the SILL program is indeed the highest among all possible inputs $[\![K_1, \dots, K_n]\!]$. □

Finally, Thm. 5.9 states the relative completeness of the algorithm.

Theorem 5.9 (Relative completeness of worst-case input generation)**.** *Given a SILL program and a collection* K *of resource-annotated session skeletons, assume the following:*

(A1) *The processes are typable in resource-aware SILL.*

(A2) *The cost bound of* K *is tight on some finitely long execution path of the SILL program.*

(A3) *The background theory for path constraints is decidable.*

Then the worst-case input generation algorithm Alg. 1 returns a valid worst-case input.

Proof. The assumption A1 is necessary because session skeletons, which define the shape of a worst-case input to be synthesized and guide the symbolic execution, are based on resource-annotated session types. Hence, all channels in the SILL program must be typable with resource-annotated session types.

Under the assumption A2, there exists some finitely long execution path π in the SILL program that has the same high-water mark as the cost bound encoded in the resource-annotated skeletons K. The symbolic execution exhaustively searches for an execution path where the cost bound is tight, until a suitable execution path is found or all execution paths fail (Alg. 1). Hence, we will eventually find the target execution path π, thanks to Prop. 5.7 stating the correctness of tracking the cost during the symbolic execution. The symbolic execution has a risk of non-termination on some execution paths. To avoid being stuck in the exploration of one execution path, the symbolic execution can explore multiple execution paths in parallel such that any finite execution path will eventually be explored. □

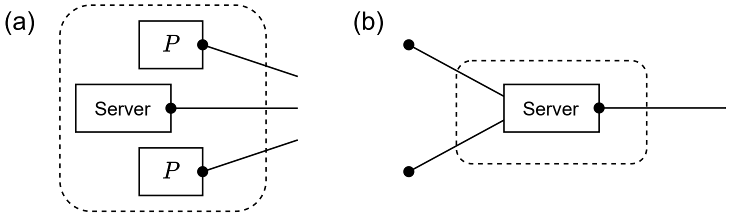

FIGURE 11. (a) First model of a web server. Here, an independent process P is assigned to each browser's session that wants to communicate with the server. (b) Second model of a web server where a scheduler is modeled.

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 session type of the provided channel is

$$c : A \otimes A \otimes \mathbf{1}. \tag{6.1}$$

That is, the server sequentially spawns a new channel of session type A for each of the two browsers. The server after spawning two channels is depicted in Fig. 11 (a). The channel providers P of these channels run independently of each other—the server cannot control the order of events on these channels.

Resource-annotated session type A is defined as

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

In the session type 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 a label ack followed by an integer (ideally $1+y$) or a label timeout to indicate that the browser is inactive.

After successful completion of the handshake, the server sends a label 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 a label 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 $\mathsf{success}$ branch has further communication. The detailed implementation of such a server is given in Eq (D.2) (Appendix D).

Let a session skeleton for the external channel c be $K_1 \otimes K_2 \otimes \mathbf{1}$, where the session skeleton K_i $(i = 1, 2)$ is

$$K_i \coloneqq z_{i,1} \supset \triangleleft^{1}(\mathsf{int} \times \mathsf{int}) \wedge \&\{\mathsf{ack} : z_{i,2} \supset \oplus\{\mathsf{success} : \triangleleft^{1}\mathbf{1}, \mathsf{failure} : \mathbf{1}\}, \mathsf{timeout} : \mathbf{1}\}. \tag{6.3}$$

Here, $z_{i,1}$ and $z_{i,2}$ are skeleton variables of the integer type. Thus, the skeleton $K_1 \otimes K_2 \otimes \mathbf{1}$ is simply obtained from the original session type $A \otimes A \otimes \mathbf{1}$ by substituting fresh skeleton variables for all occurrences of int in the session type.

The worst-case cost bound of the skeleton $K_1 \otimes K_2 \otimes \mathbf{1}$ is 4. This is because each K_i $(i = 1, 2)$ has a worst-case cost bound of 2, which happens if the browser sends the label ack to the server. The worst-case input generation algorithm generates a worst-case input

$$z_{1,1} = x_1 \quad z_{1,2} = y_1 + 1 \quad z_{2,1} = x_2 \quad z_{2,2} = y_2 + 1, \tag{6.4}$$

where $x_1, x_2 \in \mathbb{Z}$ are unconstrained, and $y_1, y_2 \in \mathbb{Z}$ are the integers sent back from the server to a browser. The 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 SILL implementation does not faithfully model the true DoS attack. A DoS attack creates a large number of so-called half-open sessions, where the completion of the handshake is delayed by withholding the label ack. Meanwhile, our SILL implementation does not model the withholding of the label ack. Instead, all browsers' sessions will eventually terminate in our SILl implementation, and the worst-case high-water-mark cost of the entire concurrent program happens when the high-water marks of all individual processes coincide. To faithfully model a DoS attack, it would be necessary to model the withholding of sending the label ack in a large number browsers' sessions at the same. However, it cannot be faithfully modeled in SILL due to the following limitations in the expressivity: (i) SILL cannot withhold events (e.g., sending and receiving messages)—they must proceed; and (ii) SILL cannot specify the exact timing of events, such as that certain events on different channels and processes all happen at the same time.

6.2. **Coordinating Sessions with Schedulers.** Now consider an alternative implementation where it is the external world that spawns channels. The typing judgment of the channel c is

$$c : A \multimap A \multimap \mathbf{1}, \tag{6.5}$$

where the session type A (without resource annotations) is

$$A \coloneqq \mathsf{int} \wedge (\mathsf{int} \times \mathsf{int}) \supset \oplus\{\mathsf{ack} : \mathsf{int} \wedge \&\{\mathsf{success} : \mathbf{1}, \mathsf{failure} : \mathbf{1}\}, \mathsf{timeout} : \mathbf{1}\}. \tag{6.6}$$

The resource annotation in the session type A 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 (Fig. 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 typing judgment of channel c is

$$c : A_{\text{anno}} \multimap A \multimap \mathbf{1}, \tag{6.7}$$

where the resource-annotated session type for the first browser is

$$A_{\text{anno}} \coloneqq \mathsf{int} \wedge \triangleright^{1} (\mathsf{int} \times \mathsf{int}) \supset \oplus\{\mathsf{ack} : \mathsf{int} \wedge \&\{\mathsf{success} : \triangleright^{1}\mathbf{1}, \mathsf{failure} : \mathbf{1}\}, \mathsf{timeout} : \mathbf{1}\}. \tag{6.8}$$

and the session type A for the second browser, which happens to require zero potential, is given in Eq (6.6). 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, pages 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, pages 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.

[LSS93] Patrick Lincoln, Andre Scedrov, and Natarajan Shankar. Linearizing intuitionistic implication. *Annals of Pure and Applied Logic*, 60(2):151–177, 1993. URL: https://www.sciencedirect.com/science/article/pii/016800729390041B, doi:10.1016/0168-0072(93)90041-B.

[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.

[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, pages 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.

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\mathsf{msg}(c_1, w_1, c_1 \leftarrow c_2)\\ \mathsf{proc}(d, w_2, P)\end{array}}{\mathsf{proc}(d, w_1 + w_2, P[c_2/c_1])}\mathsf{fwd}_r^-$$

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

Appendix A. Resource-Aware SILL

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

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.

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

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

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

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

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

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

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

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

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

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

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.

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

Appendix B. Session Skeletons

B.1. **Checking Finiteness of Input Portions.** The SMT-LIB2 encoding of Eq (4.3) is displayed below.

```
(set-logic UFLIA)
(declare-fun f (Int) Bool)
(assert (forall ((x Int))
```

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

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

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

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

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

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

Figure 14. Number of input actions in a skeleton.

```
    (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 Fig. 14. 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.

B.2. **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.

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

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

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

The issue illustrated by Eq (4.6) 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 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.

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 multiset of predicates $\mathsf{msg}(c, M)$. Given a channel $c : K$ (where K is a skeleton), let $\text{inputs}(K, c)$ denote the set of possible inputs from the perspective of c's provider. It is defined in Fig. 15. The dual $\text{outputs}(K, c)$ is the set of all outputs for c's provider. Since $\text{outputs}(\cdot,\cdot)$ is defined similarly, we omit its definition.

Definition C.1 (Set of possible 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 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

$$[\![K_1, \ldots, K_n]\!] \coloneqq \prod_{1 \leq i \leq m} \text{inputs}(K_i, c_i) \times \prod_{m < i \leq n} \text{outputs}(K_i, c_i). \tag{C.1}$$

Definition C.2 (Worst-case inputs)**.** A worst-case input $(t_1, \ldots, t_n) \in [\![K_1, \ldots, K_n]\!]$ is such that, if we add $\bigcup_{1 \leq i \leq n} t_i$ to the initial configuration of the network and run it, we obtain the highest high-water-mark cost of all possible inputs from $[\![K_1, \ldots, K_n]\!]$.

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

Figure 15. Set of possible inputs of a skeleton. The dual outputs(K, c) is the set of all outputs for c's provider.

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

Proposition C.3 (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 the 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 the 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 the 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 the 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 the 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 5.3. *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 Prop. C.3 that red potential is entirely consumed if the symbolic execution successfully terminates. Also, because the symbolic execution properly tracks blue potential, there should be no path from unconsumed blue potential to red potential in Fig. 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 Fig. 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 Fig. 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 Fig. 9 is entirely consumed by the time t_1 and t_2. Thus, all potential in Fig. 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.5 (Similarity between configurations)**.** Fix S to be a solution to a final path constraint generated once the symbolic execution terminates. Let t be an input (i.e., a multiset of predicates $\mathsf{msg}(c, M)$) induced by the solution S to the final path constraint. Consider some configuration C_{sym} during the symbolic execution and a configuration C_{cost} for the cost semantics. The similarity relation $C_{\text{sym}} \sim C_{\text{cost}}$ holds if and only if there is an injection from the multiset $t \cup C_{\text{sym}}$ of predicates to the multiset C_{cost} of predicates such that each pair in the injection satisfies the similarity relation (Defn. 5.6).

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

Theorem 5.7. *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 the cost semantics. These configurations satisfy two conditions: (i) $C_{1,sym}$ transitions to $C_{2,sym}$ in one step of the symbolic execution and (ii) $C_{1,sym} \sim C_{1,cost}$ holds. Then there exists a configuration $C_{2,cost}$ of the cost semantics such that the following diagram commutes:*

$$\begin{tikzcd} C_{1,sym} \arrow[r, "w"'] \arrow[d, no head, squiggly] & C_{2,sym} \arrow[d, no head, squiggly] \\ C_{1,cost} \arrow[r, "w"', "\leq 1"] & C_{2,cost} \end{tikzcd} \tag{C.2}$$

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 Defn. 5.5). 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 the 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 the symbolic execution and 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 the symbolic execution. □

Proposition C.7 (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 the 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 the symbolic execution such that the following diagram commutes:*

$$\begin{tikzcd}
C_{1,sym} \arrow[r, "\geq 1", "w"'] \arrow[d, squiggly, dash] & C_{2,sym} \arrow[d, squiggly, dash] \\
C_{1,cost} \arrow[r, "w"'] & C_{2,cost}
\end{tikzcd} \tag{C.3}$$

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 (Fig. 6). In the 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 Prop. 5.7, we save potential somewhere instead of throwing it away. □

C.4. Symbolic Execution.

Some of the key rules for the symbolic execution are already presented in Fig. 10. The remaining key rules are given in Figs. 16 and 17. Figs. 10, 16 and 17 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 the symbolic execution, if we choose to explore the first branch, we add b to a path constraint. As the symbolic execution for the functional layer is already presented in a prior work [WH19], this article omits it.

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

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

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

$$\dfrac{\begin{array}{c}\textsf{proc}(\Delta, c_1 : A_1 \multimap A_2, c_2 : A_1; p \vdash \textsf{send}\ c_1\ c_2; P :: (d : D), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \\ c_1' \text{ is fresh}\end{array}}{\begin{array}{c}\textsf{proc}(\Delta, c_1' : A_2; p \vdash P[c_1'/c_1] :: (d : D), \phi, (\mathrm{IDs}_s, \mathrm{IDs}_p)) \\ \textsf{msg}(c_1', \textsf{send}\ c_1\ c_2; c_1' \leftarrow c_1, \top, (\mathrm{IDs}_s, \emptyset))\end{array}}\multimap\! S$$

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

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

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

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

FIGURE 16. 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

$$d_1 \leftarrow P \leftarrow \cdot; \textsf{send}\ c\ d_1; d_2 \leftarrow P \leftarrow \cdot; \textsf{send}\ c\ d_2; \textsf{close}\ c, \tag{D.1}$$

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

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

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

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

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

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

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

FIGURE 17. Remaining key rules in the symbolic execution for the process layer (part 2).

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

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

Integers x and y are called sequence numbers and are stored in the server. This is why we have $\mathsf{tick}\ 1$. Additionally, once the handshake is completed successfully, we run $\mathsf{tick}\ 1$ 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

$$\begin{aligned} P \coloneqq{} & x_1 \leftarrow \mathsf{recv}\ d_1; \mathsf{tick}\ 1; \mathsf{send}\ d_1\ \langle x_1 + 1, y_1\rangle; \\ & \quad \mathsf{case}\ d_1\ \{\mathsf{ack} \hookrightarrow \ldots; x_2 \leftarrow \mathsf{recv}\ d_2; \mathsf{tick}\ 1; \mathsf{send}\ d_2\ \langle x_2 + 1, y_2\rangle; \\ & \qquad\qquad\qquad \mathsf{case}\ d_2\ \{\mathsf{ack} \hookrightarrow \ldots, \mathsf{timeout} \hookrightarrow \ldots\}, \\ & \qquad\quad \mathsf{timeout} \hookrightarrow \ldots\}. \end{aligned} \tag{D.3}$$

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

$$d_1 \leftarrow \mathsf{recv}\ c; d_2 \leftarrow \mathsf{recv}\ c; c \leftarrow P \leftarrow d_1, d_2, \tag{D.4}$$

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

$$\begin{aligned} P \coloneqq\ & x_1 \leftarrow \mathsf{recv}\ d_1; x_2 \leftarrow \mathsf{recv}\ d_2; \\ & \mathsf{tick}\ 1; \mathsf{tick}\ 1; \mathsf{send}\ d_1\ \langle x_1 + 1, y_1 \rangle; \mathsf{send}\ d_2\ \langle x_2 + 1, y_2 \rangle \\ & \mathsf{case}\ d_1\ \{\mathsf{ack} \hookrightarrow \mathsf{case}\ d_2\ \{\ldots\}, \mathsf{timeout} \hookrightarrow \mathsf{case}\ d_2\ \{\ldots\}\}. \end{aligned} \tag{D.5}$$

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

$$A_{\text{anno}} \coloneqq \mathsf{int} \wedge \triangleright^{1}(\mathsf{int} \times \mathsf{int}) \supset \oplus\{\mathsf{ack} : \mathsf{int} \wedge \&\{\mathsf{success} : \triangleright^{1}\mathbf{1}, \mathsf{failure} : \mathbf{1}\}, \mathsf{timeout} : \mathbf{1}\}. \tag{D.6}$$

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.