Stuttering for Free

MINKI CHO*, Seoul National University, Korea
YOUNGJU SONG*, MPI-SWS, Germany
DONGJAE LEE, Seoul National University, Korea
LENNARD GÄHER, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany

One of the most common tools for proving behavioral refinements between transition systems is the method of *simulation* proofs, which has been explored extensively over the past several decades. Stuttering simulations are an extension of traditional simulations—used, for example, in CompCert—in which either the source or target of the simulation is permitted to "stutter" (stay in place) while the other side steps forward. In the interest of ensuring soundness, however, existing stuttering simulations restrict proofs to only perform a finite number of stuttering steps before making *synchronous progress*—a step of reasoning in which both sides of the simulation progress forward together. This restriction guarantees that a terminating program cannot be proven to simulate a non-terminating one.

In this paper, we observe that the requirement to eventually achieve synchronous progress is burdensome and, what's more, unnecessary: it is possible to ensure soundness of stuttering simulations while only requiring *asynchronous progress* (progress on both sides of the simulation that may be achieved with only stuttering steps). Building on this observation, we develop a new simulation technique we call **FreeSim** (short for "freely-stuttering simulations"), mechanized in Coq, and we demonstrate its effectiveness on a range of interesting case studies. These include a simplification of the meta-theory of CompCert, as well as the **DTrees** library, which enriches the ITrees (Interaction Trees) library with dual non-determinism.

CCS Concepts: • **Theory of computation** → **Logic**; *Automata over infinite objects.*

Additional Key Words and Phrases: stuttering simulation, Coq, verification

ACM Reference Format:
Minki Cho, Youngju Song, Dongjae Lee, Lennard Gäher, and Derek Dreyer. 2023. Stuttering for Free. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 281 (October 2023), 28 pages. https://doi.org/10.1145/3622857

1 INTRODUCTION

Behavioral refinement is a widely-used notion in formal verification for relating two transition systems. For a given notion of behavior—*e.g.*, (non-)termination, a trace of visible events—a target system behaviorally refines a source system if the set of behaviors of the target is included in the set of behaviors of the source. For example, the correctness of a compiler is usually stated as a behavioral refinement between the source program and the compiled target program, and indeed verified compilers such as CompCert [Leroy 2006] establish such a result formally. Refinement can also be used as a technique for program verification; for instance, the verification frameworks [Gu

*Minki Cho and Youngju Song contributed equally to this work and should be considered joint first authors.

Authors' addresses: Minki Cho, Seoul National University, Korea, minki.cho@sf.snu.ac.kr; Youngju Song, MPI-SWS, Saarland Informatics Campus, Germany, youngju@mpi-sws.org; Dongjae Lee, Seoul National University, Korea, dongjae.lee@sf.snu.ac.kr; Lennard Gäher, MPI-SWS, Saarland Informatics Campus, Germany, gaeher@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

2475-1421/2023/10-ART281
https://doi.org/10.1145/3622857

```
Endt[a,b] := print(a).print(b).stop

Runt[a,b] := τ.Endt[a,b] +
             τ.GoPickB[0] + τ.GoPickB[1]
GoPickB[a] := τ.Runt[a,0] + τ.Runt[a,1]

INITIAL: Runt[0,0]
```

```
Ends[a,b] := print(a).print(b).stop

Runs[a,b] := τ.Ends[a,b] +
             τ.GoPickA[0] + τ.GoPickA[1]
GoPickA[b] := τ.Runs[0,b] + τ.Runs[1,b]

INITIAL: Runs[0,0]
```

Fig. 1. An example where asynchronous execution is useful.

et al. 2015; Lorch et al. 2020; Sammler et al. 2019; Turon et al. 2013] establish program correctness by showing that the program in question refines a more abstract mathematical specification formulated as a state transition system (STS).

One of the most common tools for proving behavioral refinements is the method of *simulations* [Milner 1971]. In its simplest form, the simulation technique requires us to (i) specify a relation $t \precsim s$ on source states s and target states t (called the *simulation relation*), (ii) ensure that the initial states of the source and target systems are related by the simulation relation, and (iii) prove that for any given pair of related states $t \precsim s$, if the target t can execute a step to t', then the source s can simulate this behavior by stepping to s', such that again $t' \precsim s'$, while emitting the same visible events. In this way, simulations offer a way to prove behavioral refinement—a *global* property about the entire traces of execution of the two programs—in a *temporally local* fashion.

Going beyond the basic simulation technique, many extensions have been proposed with the aim of making simulation proofs simpler and easier to carry out. One example is the large class of "up-to" techniques, which weaken the proof obligation in condition (iii) above so that t' and s' need not be related by the simulation relation itself but rather by some larger relation [Pous 2016].

In this paper, we focus on a different but also commonly used extension of the simulation technique, namely the ability to "stutter". Specifically, *stuttering simulations* [Namjoshi 1997] loosen the requirement in condition (iii) that the steps of the source and target programs must be *synchronous*—*i.e.*, matched up one-to-one. Instead, stuttering simulations support what we call *asynchronous reasoning*: so long as either the source or target program is only making a "τ-step" (*i.e.*, a step that does not emit a visible event), the other program is allowed to "stutter" (*i.e.*, stay in its current state and *not* take a step). This flexibility has been shown to come in very handy in many proofs [Leroy 2006].

Of course, such stuttering cannot be allowed without any restrictions: if a proof were allowed to take infinitely many stuttering steps in a row, it would be trivially easy to prove that a terminating program is in simulation with a program that makes an infinite sequence of τ-steps. Hence, to ensure soundness of the simulation method, stuttering simulations somehow have to ensure that the proof argument only stutters *finitely* before eventually taking a step of *synchronous progress*: a reasoning step, à la condition (iii), where both source and target programs progress forward together. As we will see in §2, there are multiple ways to ensure this, either *explicitly* (using a well-founded "stutter budget" that decreases at each stuttering step) or *implicitly* (using a mixed inductive-coinductive definition to ensure termination of stuttering).

The need for asynchronous progress. In this paper, we observe that the requirement to eventually (after finite stuttering) achieve *synchronous progress* is burdensome and, what's more, unnecessary: it is possible to ensure soundness of stuttering simulations without requiring eventual synchronous progress. In particular, we propose a new proof technique that we call *freely stuttering simulations*, which relaxes the condition of synchronous progress to one of *asynchronous progress*.

To motivate asynchronous progress, let us consider a concrete example: the two transition systems shown in Fig. 1, expressed in CCS [Milner 1989]. At a high level, both systems nondeterministically pick two boolean numbers, `a` and `b`, using CCS's choice operator `p + q`. Initially, `a` and `b` are initialized to zero. After choosing `a` and `b`, the systems then nondeterministically either print the numbers (in the subprocess `End[a,b]`), or choose new numbers.

The two systems only differ in the order in which they choose `a` and `b`: the left system first picks `a`, before picking `b` in the subprocess `GoPickB`. On the other hand, the right system first picks `b`, before picking `a` in the subprocess `GoPickA`. Intuitively, both systems are behaviorally equivalent: the order in which `a` and `b` are picked is not part of the system's visible behavior, as only silent τ actions are taken—only the final `print` events are visible.

When proving a simulation between these two systems, existing simulations unnecessarily restrict the proof strategy. To explain the limitation, we focus on the key part of proving the simulation, starting with the simulation goal $\texttt{Run}_\texttt{t}\texttt{[a,b]} \precsim \texttt{Run}_\texttt{s}\texttt{[a,b]}$. In the case that the system does not terminate and continues execution (`GoPickB[0] + GoPickB[1]`), we should prove that we can come back to the original goal ($\texttt{Run}_\texttt{t}\texttt{[a,b]} \precsim \texttt{Run}_\texttt{s}\texttt{[a,b]}$) by executing the target (left) and source (right) systems, and then conclude the proof by coinduction. Ideally, the verification should work independently of the order in which we execute the target and the source systems. In particular, due to the non-deterministic choices in this system, we would like to be able to execute the target and source sides *asynchronously*: that is, to first execute the target until it reaches $\texttt{Run}_\texttt{t}\texttt{[a,b]}$ in order to witness the values picked for `a` and `b`, and only after that execute the source.

However, such a verification is **not** possible in existing simulations! The reason is simple: existing simulations require the proof to eventually make synchronous progress, which is achieved when the source and the target systems take a synchronous step but not when they execute stuttering steps asynchronously! As a result, the only way to prove this example in existing simulations is to (1) execute one stuttering step in the target, (2) execute one synchronous step (both in the target and in the source), and then (3) execute one stuttering step in the source.[1] This is clearly fiddly and inconvenient: it means that the user needs to understand the system in detail before starting the verification in order to execute a synchronous step at just the right point.

Introducing: freely-stuttering simulations. In this paper, we propose a new approach to ensuring soundness of stuttering simulations, which we dub **FreeSim** (short for "freely-stuttering simulations").

With FreeSim, we lift the rigid restrictions of existing simulations and allow proofs to stutter freely so long as they eventually make *asynchronous progress*. In FreeSim's simplest form, the simulation is equipped with two boolean flags, denoting whether the target and source programs have taken a step, respectively. Executing a step in the target sets the target flag to `true`, while executing a step in the source does the same for the source flag. Finally, when applying a coinductive hypothesis (*e.g.*, when reaching `Run[a,b]` on both sides in the above example), FreeSim ensures that *asynchronous progress* has been made by requiring both boolean flags to be set to true. This prevents unsound circular reasoning much like the requirement for synchronous progress in previous simulations, but—crucially—it allows progress to be achieved even when the target and the source steps are made completely independently (as stuttering steps). For example, in the context of our motivating example, FreeSim allows us to write the proof the way we wanted: the target can first freely execute stuttering steps until it reaches $\texttt{Run}_\texttt{t}\texttt{[a,b]}$, then the source freely executes stuttering steps until it reaches $\texttt{Run}_\texttt{s}\texttt{[a,b]}$, and since each side independently took at least one step, asynchronous progress has been achieved, and we can conclude the proof.

[1] "Executing a stuttering step" is a shorthand for saying "executing a (τ-)step while the counterpart stutters".

In addition, FreeSim is designed to subsume *both* explicit and implicit stuttering: the proofs enabled by simulations with implicit and explicit stuttering can be exactly *replayed* in simulations designed with FreeSim. This in turn means that FreeSim enjoys the complementary benefits of both, enabling new kinds of proofs that combine the benefits of both sides.

We emphasize that it is not our goal to come up with a simulation that has additional proof power over existing ones: indeed, we prove that simulations with explicit and implicit stuttering are equivalent to FreeSim. However, FreeSim provides a better interface to the user, allowing more *flexibility* in completing the proof.

Enabling metatheoretic results. A further major benefit of FreeSim is that it allows one to simplify the simulation's definition, making the proof of meta-theoretic results about the simulation much more feasible. This holds especially true once our simulation needs to handle more primitives for more complex systems. As a case in point, we consider *Interaction Trees* [Xia et al. 2019] (which can be seen as state transition systems with additional structure, thus enjoying more interesting metaproperties) with *dual non-determinism* (both demonic and the angelic non-determinism). As a result, we provide the first equational theory for Interaction Trees with dual non-determinism. The proofs involve non-trivial induction-coinduction techniques, and our simplified simulation relation was indeed the key enabler for this result.

Contributions. This paper develops the idea of **FreeSim**, making the following contributions:

- FreeSim: we propose a novel stuttering simulation allowing fully asynchronous reasoning (§3).
- We show that while FreeSim is propositionally equivalent to existing stuttering simulations (§5), it strictly subsumes existing ones in terms of user interface (§3.3). To formally capture the idea of such a "subsumption" in user interface, we devise a novel notion dubbed **replayability**.
- We show that FreeSim allows new forms of reasoning via stronger reasoning principles (§3.2).
- We show that FreeSim supports simpler definitions, especially when adding new primitives (§3.4 and §5).
- We formalize FreeSim and prove its key properties including adequacy (§4).
- We showcase the utility of FreeSim with two case studies: we use it to simplify the meta-theory of CompCert, and develop the **DTrees** library, which is an extension of the ITrees (Interaction Trees) library enriched with dual non-determinism (§6).

All our results are formalized in Coq and available in the accompanying supplementary material [Cho et al. 2023].

2 BACKGROUND

Before we begin, we first setup a few basic notations and definitions for the rest of the paper. The simulation relations we discuss in this paper are centered around STS, which are defined as follows:

$$\{X \in \mathbf{Set},\ x_0 \xhookrightarrow{e} x_1 \in \mathcal{P}(X \times \mathcal{E} \times X),\ \mathrm{init} \in X,\ \mathrm{sort} \in X \to \mathrm{Sort}\} \qquad \text{(State Transition System)}$$
$$\mathrm{Sort} \triangleq \{\mathrm{Tau}\} \uplus \{\mathrm{Vis}\} \uplus \{\mathrm{Ret(r)}\}$$

X is a set of states, $x_0 \xhookrightarrow{e} x_1$ denotes a transition from a state x_0 to x_1 emitting an event e (which could be a silent event, τ), and "init" is the initial state. Additionally, without loss of generality, we assume that each state is classified (by the "sort" component) as one of the sorts Tau, Vis, or Ret. Ret states are final and do not have any outgoing transitions. Tau states can have an arbitrary number of outgoing transitions, but all of them should be silent (τ). Conversely, Vis states can have at most one outgoing transition, and this transition is observable. Any STS can be normalized (with

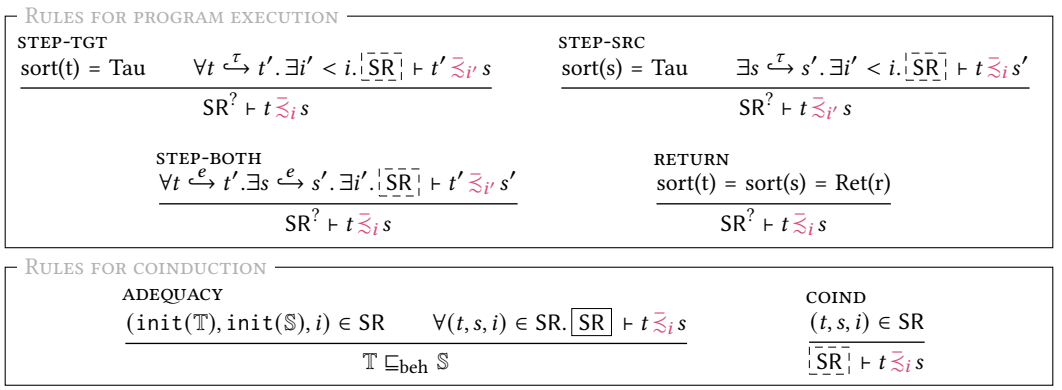


Fig. 2. Interface of the explicit stuttering simulation.

minor local adjustments) to a (behaviorally) equivalent STS satisfying these axioms around Sort. We assume these axioms to simplify our presentation.

In the rest of the section, we discuss existing definitions for stuttering simulations on STS, *i.e.*, relations that relate a target and source STS. We start by explaining the "interfaces" (in terms of proof rules) of the existing techniques with a focus on how they enforce finite stuttering: we discuss *explicit/implicit* stuttering simulations (ESim/ISim) in §2.1/§2.2. While ESim and ISim both have a sophisticated coinductive definition, their high level user interfaces are easy to understand. Finally, we revisit the example from Fig. 1 to understand their limitations (§2.3).

2.1 Explicit Stuttering Simulation

The interface for the ESim (using notation $\bar{\precsim}$) is given in Fig. 2. With these rules, a simulation proof proceeds as follows. It begins by applying ADEQUACY with the simulation relation SR of choice. SR is a ternary relation between target states, source states, and "stuttering indices" $\mathbb{I}$ equipped with a well-founded order $<$. Then, one is left with proving $\boxed{\mathrm{SR}} \vdash t \mathrel{\bar{\precsim}_i} s$ for each triple $(t, s, i) \in \mathrm{SR}$ where the boxed SR on the left denotes *guarded* coinductive hypotheses. Each proof proceeds by applying the rules for program execution (upper box in Fig. 2) and can be concluded by either reaching a final state (RETURN) or appealing to the coinductive hypothesis (COIND). For the latter to work, coinductive hypotheses should have been *unguarded* as denoted by the dashed-box notation. Coinductive hypotheses get unguarded whenever a program execution rule is applied.

Now we explain the program execution rules, consider how coinductive progress is made, and see where the stuttering index comes in. First and foremost, the STEP-BOTH executes a target step ($\forall t \xrightarrow{e} t'$), finds a matching step in the source ($\exists s \xrightarrow{e} s'$), and proceeds with the proof with a fresh stuttering index i' (we will see its use soon). The rule is applicable regardless of whether coinductive hypotheses are guarded or not, as noted by the question mark ($R^?$), and the rule unguards the coinductive hypotheses ($\boxed{\boxed{\mathrm{SR}}} \vdash t' \mathrel{\bar{\precsim}_{i'}} s'$).

Since this rule alone cannot relate STS that do not perfectly line up, there are rules for asynchronous execution. Consider the following example where $\sqsubseteq_{\mathrm{beh}}$ denotes behavioral refinement:

$$L \triangleq \quad \mathbf{loop}\,\{\,\mathbf{skip}; \mathit{print}(42)\,\} \sqsubseteq_{\mathrm{beh}} \mathbf{loop}\,\{\,\mathit{print}(42)\,\} \quad \triangleq R \qquad \text{(SOUND)}$$

One should be able to verify the above by utilizing the STEP-TGT rule which would only execute the target step **skip** on the left side. However, one should *not* be able to verify the following:

$$\mathbf{loop}\,\{\,\mathbf{loop}\,\{\,\mathbf{skip}\,\}; \mathit{print}(42)\,\} \sqsubseteq_{\mathrm{beh}} \mathbf{loop}\,\{\,\mathit{print}(42)\,\} \qquad \text{(UNSOUND)}$$

because the target has a different behavior (silent divergence) from the source (infinitely printing "42").

This is precisely what the stuttering index ensures. The stuttering index i allows one to execute t and s (STEP-TGT and STEP-SRC) independently of each other for at most i steps. For instance, SOUND could be verified as follows:

$$
\begin{array}{lll}
& L \sqsubseteq_{\text{beh}} R & \\
\Longleftarrow & \boxed{\{(L, R, 1)\}} \vdash L \,\bar{\precsim}_1\, R & \text{by ADEQUACY} \\
\Longleftrightarrow & \boxed{\{(L, R, 1)\}} \vdash \textbf{skip}; \mathit{print}(42); L \,\bar{\precsim}_1\, \mathit{print}(42); R & \text{by unfolding} \\
\Longleftarrow & \boxed{\{(L, R, 1)\}} \vdash \mathit{print}(42); L \,\bar{\precsim}_0\, \mathit{print}(42); R & \text{by STEP-TGT} \\
\Longleftarrow & \boxed{\{(L, R, 1)\}} \vdash L \,\bar{\precsim}_1\, R & \text{by STEP-BOTH} \\
\Longleftarrow & (L, R, 1) \in \{(L, R, 1)\} & \text{by COIND}
\end{array}
$$

Note that STEP-BOTH refreshes the index to 1, which is needed to conclude the proof with COIND.

Moreover, it is easy to see that one indeed *cannot* verify UNSOUND with ESim: no matter which stuttering index one chooses, one can apply STEP-TGT only a finite number of times (since i is an element of a well-founded order) and cannot execute infinite loops like **loop**{ **skip** } with this.

Annoyance with managing index. One big disadvantage of the explicit stuttering interface is that the user of the proof rules has to reason about indices for every step of execution. Even worse, every time a synchronous step is performed with STEP-BOTH, the new index has to be carefully chosen large enough to last until the next synchronous step. To see it more clearly, consider the following example adapted from a tutorial in ITrees [Xia et al. 2019]:

$$
L \triangleq \quad \textbf{loop}\,\{\,\mathit{print}(\texttt{fib(n)})\,\} \qquad \sqsubseteq_{\text{beh}} \qquad \textbf{loop}\,\{\,\mathit{print}(\mathbb{FIB}(\texttt{n}))\,\} \quad \triangleq R \tag{FIB}
$$

For an arbitrary natural number `n`, both the target (left) and the source (right) print the n-th Fibonacci number in an infinite loop. In the source program, $\mathbb{FIB}(n)$ is a mathematical value representing the n-th Fibonacci number, while in the target program, `fib` is itself implemented in the programming language as follows:

```
def fib(n) ≡ if n >= 2 then fib(n-1) + fib(n-2) else 1
```

That is, the refinement is abstracting a more concrete program into a more abstract program.

Now we consider the verification of FIB using ESim. For this, we need to know *upfront* how many reduction steps the target will take in order to set up a sufficient stuttering index. This clearly is an inconvenience especially given that making the right choice requires semantic understanding of the program. Specifically, a minimal stuttering index for the above is defined recursively as follows:

$$
\mathrm{I}_n = 2 + \mathrm{I}_{n-1} + \mathrm{I}_{n-2} \;\; (\text{when } n \geq 2) \qquad \&\& \qquad \mathrm{I}_n = 2 \;\; (\text{otherwise})
$$

where the number 2 in both cases corresponds to taking one step each for (i) evaluating the condition of the **if** statement, and (ii) evaluating the **if** statement itself. Note that I_n is not simply the same as $\mathbb{FIB}(n)$ and one needs semantic understanding of the program *upfront* to come up with such a construction. With this choice, FIB could be verified with $\mathrm{SR} \triangleq \{(L, R, \mathrm{I}_n)\}$.

2.2 Implicit Stuttering Simulation

Implicit stuttering simulation (ISim) circumvents the inconvenience of ESim by completely removing indices from the simulation interface. Rules for the ISim (using notation $\precsim$) are presented in Fig. 3.

Of course, completely removing the stuttering index has repercussions: as demonstrated by UNSOUND, simply removing the stuttering index from the proof rules would lead to an unsound proof system. Thus, in ISim, the rules STEP-SRC and STEP-TGT do *not* unguard the coinductive hypotheses anymore (*i.e.*, $\boxed{\mathrm{SR}}$ and $\boxed{\mathrm{SR}}$ remain unchanged). In an attempt to prove UNSOUND using ISim, this

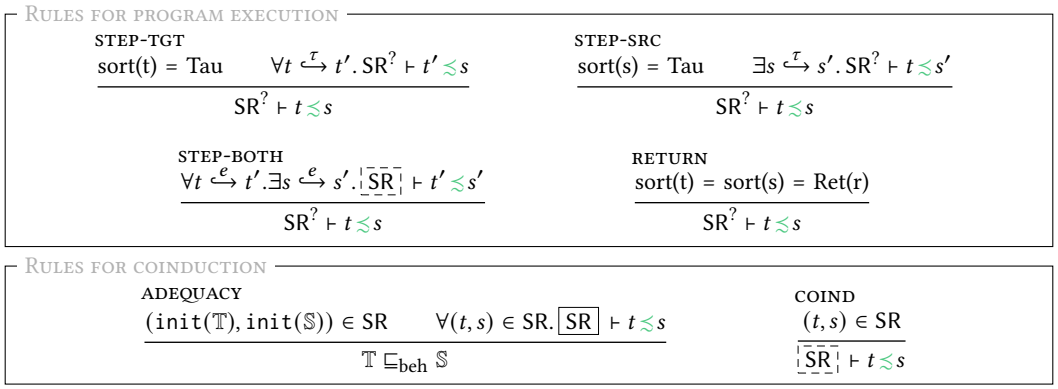


Fig. 3. Interface of the implicit stuttering simulation.

would be reflected by the coinductive hypotheses never getting unguarded, and thus concluding the proof by COIND would be rightly forbidden.

Now, with ISim, FIB could be verified much more easily as follows:

$$
\begin{array}{lll}
& L \sqsubseteq_{\text{beh}} R & \\
\Longleftarrow & \boxed{\{(L,R)\}} \vdash L \precsim R & \text{by ADEQUACY} \\
\Longleftrightarrow & \boxed{\{(L,R)\}} \vdash \mathit{print}(\texttt{fib(n)}); L \precsim \mathit{print}(\mathbb{FIB}(\texttt{n})); R & \text{by unfolding} \\
\Longleftarrow & \boxed{\{(L,R)\}} \vdash \mathit{print}(\mathbb{FIB}(\texttt{n})); L \precsim \mathit{print}(\mathbb{FIB}(\texttt{n})); R & \text{by repeating STEP-TGT} \\
\Longleftarrow & \boxed{\{(L,R)\}} \vdash L \precsim R & \text{by STEP-BOTH} \\
\Longleftarrow & (L,R) \in \{(L,R)\} & \text{by COIND} \quad \square
\end{array}
$$

First, we apply ADEQUACY to turn the behavioral refinement into a simulation proof. Then, we apply STEP-TGT repeatedly to evaluate `fib(n)` in the target to the value $\mathbb{FIB}(\texttt{n})$. Now, we can execute $\mathit{print}(\mathbb{FIB}(\texttt{n}))$ synchronously on both sides with STEP-BOTH, in the process unguarding the coinductive hypothesis. Finally, we conclude the proof with COIND using unguarded hypothesis.

While ESim and ISim have different interfaces, they share the same crux:

To conclude the proof by coming back to the original proof state,
one must have executed STEP-BOTH *somewhere before.*

2.3 Problem Revisited

With this observation, let us revisit the problematic example from the introduction (Fig. 1). Using ISim (similar arguments apply to ESim), we would like to conduct the following proof:

$$
\begin{array}{lll}
& \texttt{Run}_t\texttt{[a,b]} \sqsubseteq_{\text{beh}} \texttt{Run}_s\texttt{[a,b]} & \\
\Longleftarrow & \boxed{\bigcup_{\texttt{a}\,\texttt{b} \in \{0,1\}} \{(\texttt{Run}_t\texttt{[a,b]}, \texttt{Run}_s\texttt{[a,b]})\}} \vdash \forall \texttt{a}\,\texttt{b}.\ \texttt{Run}_t\texttt{[a,b]} \precsim \texttt{Run}_s\texttt{[a,b]} & \text{by ADEQUACY} \\
\Longleftarrow & \boxed{\bigcup_{\texttt{a}\,\texttt{b} \in \{0,1\}} \{(\texttt{Run}_t\texttt{[a,b]}, \texttt{Run}_s\texttt{[a,b]})\}} \vdash \forall \texttt{a}\,\texttt{b}\,\texttt{a'}\,\texttt{b'}.\ \texttt{Run}_t\texttt{[a',b']} \precsim \texttt{Run}_s\texttt{[a,b]} & \text{by STEP-TGT (twice)} \\
\Longleftarrow & \boxed{\bigcup_{\texttt{a}\,\texttt{b} \in \{0,1\}} \{(\texttt{Run}_t\texttt{[a,b]}, \texttt{Run}_s\texttt{[a,b]})\}} \vdash \forall \texttt{a'}\,\texttt{b'}.\ \texttt{Run}_t\texttt{[a',b']} \precsim \texttt{Run}_s\texttt{[a',b']} & \text{by STEP-SRC (twice)} \\
\not\Longleftarrow & \forall \texttt{a'}\,\texttt{b'}.\ \texttt{Run}_t\texttt{[a',b']} \precsim \texttt{Run}_s\texttt{[a',b']} \in \bigcup_{\texttt{a}\,\texttt{b} \in \{0,1\}} \{(\texttt{Run}_t\texttt{[a,b]}, \texttt{Run}_s\texttt{[a,b]})\} & \text{by COIND?}
\end{array}
$$

where a trivial case for executing `End[a,b]` is omitted.

The proof starts by putting the matching `Run[a,b]` s into the coinductive hypothesis, using ADEQUACY. First, we execute the target until the end (*i.e.*, until it reaches the coinductive hypothesis) and witness the freshly picked values, `a'` and `b'`. Then, we execute the source until the end, utilizing the freshly picked values `a'` and `b'` from the target to take matching steps in the source.

However, the proof has an issue: the coinductive hypothesis never gets unguarded during the execution! Specifically, we cannot conclude the proof with COIND at the end, because the coinductive

hypothesis is still guarded. The only way out is to refactor the proof as follows:

$$
\begin{array}{lll}
& \mathtt{Run_t[a,b]} \sqsubseteq_{\mathrm{beh}} \mathtt{Run_s[a,b]} & \\
\Longleftarrow & \boxed{\bigcup_{\mathtt{a\,b}\in\{0,1\}}\{(\mathtt{Run_t[a,b]},\, \mathtt{Run_s[a,b]})\}} \vdash \forall \mathtt{a\,b.\ Run_t[a,b]} \precsim \mathtt{Run_s[a,b]} & \text{by ADEQUACY} \\
\Longleftarrow & \boxed{\bigcup_{\mathtt{a\,b}\in\{0,1\}}\{(\mathtt{Run_t[a,b]},\, \mathtt{Run_s[a,b]})\}} \vdash \forall \mathtt{a\,b\,a'.\ PickB[a']} \precsim \mathtt{Run_s[a,b]} & \text{by STEP-TGT} \\
\Longleftarrow & \boxed{\bigcup_{\mathtt{a\,b}\in\{0,1\}}\{(\mathtt{Run_t[a,b]},\, \mathtt{Run_s[a,b]})\}} \vdash \forall \mathtt{a\,b\,a'\,b'.\ Run_t[a',b']} \precsim \mathtt{PickA[b']} & \text{by STEP-BOTH} \\
\Longleftarrow & \boxed{\bigcup_{\mathtt{a\,b}\in\{0,1\}}\{(\mathtt{Run_t[a,b]},\, \mathtt{Run_s[a,b]})\}} \vdash \forall \mathtt{a'\,b'.\ Run_t[a',b']} \precsim \mathtt{Run_s[a',b']} & \text{by STEP-SRC} \\
\Longleftarrow & \mathtt{\forall a'\,b'.\ Run_t[a',b']} \precsim \mathtt{Run_s[a',b']} \in \bigcup_{\mathtt{a\,b}\in\{0,1\}}\{(\mathtt{Run_t[a,b]},\, \mathtt{Run_s[a,b]})\} & \text{by COIND} \quad \square
\end{array}
$$

Note also that yet another attempt for verification which applies the STEP-BOTH rule twice would also not work: in order to choose the correct branch in the first source step, we must know the value `b'` picked from the second target step beforehand.

To put it more generally, the problem with existing stuttering simulations is as follows. First, there are multiple ways to execute the program with the given simulation interface: given a target program step, sometimes it needs to be executed with STEP-TGT, and at other times with STEP-BOTH. We cannot always use STEP-TGT, as it does not allow us to make coinductive progress. On the other hand, we cannot always use STEP-BOTH, as this may require us to execute the source too early. Then, knowing which rule to use requires foreseeing later proof steps! Even foreseeing a single step is non-trivial when it requires a case analysis on a symbolic variable. The situation is even worse in simulations where a transition encodes mathematical conditions [Back and Wright 2012; Sammler et al. 2023; Song et al. 2023] and requires nontrivial reasoning to execute even a single step, or in the context of interactive theorem proving [The Coq Development Team 2021] where executing a step could involve nontrivial reductions in terms.

We now demonstrate FreeSim, which not only resolves this issue but also has useful properties.

3 FREELY-STUTTERING SIMULATION

In this section, we introduce our FreeSim interface and explain how it solves the aforementioned issue (§3.1). Furthermore, we show that FreeSim provides stronger compositional reasoning principles (§3.2). We also show that FreeSim subsumes both ESim and ISim, enjoying the benefits of both (§3.3). Finally, we remark on how FreeSim scales better when adding new primitives to the underlying notion of STS, using the example of dual non-determinism (§3.4).

3.1 Interface of Freely-Stuttering Simulation

Fig. 4 shows the interface of FreeSim (using notation $\overset{+}{\precsim}$). The simulation is parameterized over an index type $\mathbb{I}$ equipped with a well-founded order $<_{\mathbb{I}}$ and a greatest element $\top$. We use $\leq_{\mathbb{I}}$ to refer to the reflexive closure of $<_{\mathbb{I}}$. Now, the simulation ${}_{ps}\overset{+}{\precsim}_{pt}$ carries two indices from $\mathbb{I}$, which we call the "progress indices" for source (*ps*) and target (*pt*). For this subsection, it suffices to consider a simple well-founded order consisting of just $\bot < \top$.

Progress indices are *different* from stuttering indices: they are used to track "partial coinductive progress" made on each side of the simulation. As such, progress indices get incremented as execution goes on (thus the plus sign in the notation), while stuttering indices get decremented (thus the minus sign in the notation). Specifically, executing the target (source) will raise *pt* (*ps*) to the greatest element $\top$.

With progress indices, STEP-TGT and STEP-SRC are strengthened from ISim by now recording partial progress in *pt* and *ps*. These indices are then utilized by a new rule, COIND-PROG: it decrements both indices and in return unguards the coinductive hypotheses, *without* executing the program.

Since STEP-TGT and STEP-SRC (together with COIND-PROG) can now establish coinductive progress, we no longer need to execute two silent steps synchronously in order to constitute coinductive progress. Consequently, STEP-BOTH is now *reduced* to STEP-EVENT which is applicable only when

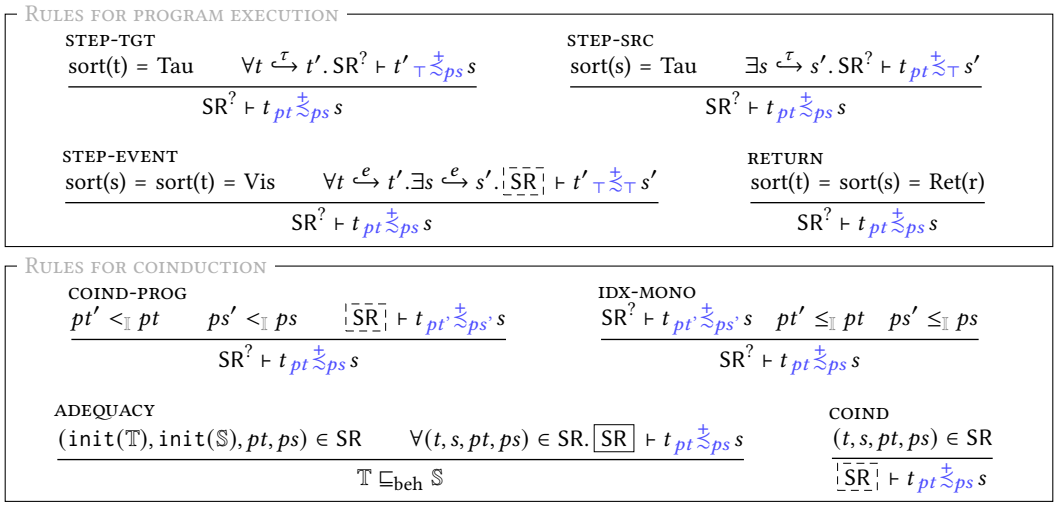


Fig. 4. Interface of freely-stuttering simulation.

both source and target are emitting a visible event (*i.e.*, are of sort Vis). In other words, STEP-BOTH could be derived from the above rules and we choose a minimal core for FreeSim.

Another way to understand progress indices are that they *decouple* program execution from coinductive reasoning (unguarding). We will see the virtue of this decoupling more clearly in §3.4.

The rest of the rules are basically the same as ISim. We additionally have a minor rule IDX-MONO that decrements progress indices, which comes in handy when proving metatheoretical results.

Problem revisited. Now, we revisit the problematic example (Fig. 1) and show how FreeSim handles it. With $\mathrm{SR} \triangleq \bigcup_{a\,b \in \{0,1\}} \{(\mathtt{Run_t[a,b]}, \mathtt{Run_s[a,b]}, \bot, \bot)\}$, the proof proceeds as follows:

$\mathtt{Run_t[a,b]} \sqsubseteq_{\mathrm{beh}} \mathtt{Run_s[a,b]}$
$\Longleftarrow \boxed{\mathrm{SR}} \vdash \forall a\,b.\ \mathtt{Run_t[a,b]} \ {}_{\bot}\overset{+}{\lesssim}_{\bot}\ \mathtt{Run_s[a,b]}$ by ADEQUACY
$\Longleftarrow \boxed{\mathrm{SR}} \vdash \forall a\,b\,a'\,b'.\ \mathtt{Run_t[a',b']} \ {}_{\top}\overset{+}{\lesssim}_{\bot}\ \mathtt{Run_s[a,b]}$ by STEP-TGT (twice)
$\Longleftarrow \boxed{\mathrm{SR}} \vdash \forall a'\,b'.\ \mathtt{Run_t[a',b']} \ {}_{\top}\overset{+}{\lesssim}_{\top}\ \mathtt{Run_s[a',b']}$ by STEP-SRC (twice)
$\Longleftarrow \boxed{\mathrm{SR}} \vdash \forall a'\,b'.\ \mathtt{Run_t[a',b']} \ {}_{\bot}\overset{+}{\lesssim}_{\bot}\ \mathtt{Run_s[a',b']}$ by COIND-PROG
$\Longleftarrow \forall a'\,b'.\ \mathtt{Run_t[a',b']} \ {}_{\bot}\overset{+}{\lesssim}_{\bot}\ \mathtt{Run_s[a',b']} \in \mathrm{SR}$ by COIND □

The proof is basically the same as the failed proof attempt in §2.3: we execute two target steps with STEP-TGT and two source steps with STEP-SRC. However, this time these stuttering steps record partial progress, turning the indices into $\top$. Thanks to this, we can apply COIND-PROG right before the COIND to unguard coinductive hypothesis, and the following COIND indeed concludes the proof.

Soundness. The intuition behind the soundness of FreeSim is (again) best understood by looking at how it prevents a proof of UNSOUND. For a given SR, consider an element with minimal *ps*. Then, since STEP-TGT cannot be applied infinitely, one must appeal to coinductive reasoning to conclude the proof. Thus, one must apply COIND-PROG, but this results in *ps* lower than the original state and one cannot conclude the proof with COIND. More details can be found in §4.4.

3.2 Stronger Compositional Reasoning

In this subsection, we show a new style of proof in FreeSim made possible with its *stronger compositional reasoning* principles. These stronger reasoning principles allow communicating partial progresses (progress indices) across different subproofs.

For expository purposes, here we consider a very simple "sequence" operator: given a pair of STS hd and tl, $hd; tl$ substitutes return states of hd—ignoring the return values—with the initial state of tl. We will consider a general composition operator $\gg\!=$ that respects return values later in §6.2. For the sequence operator, one would typically come up with the following "sequence" rule:

$$\boxed{\Gamma} \vdash hd_t \lesssim hd_s \quad \wedge \quad \boxed{\Gamma} \vdash tl_t \lesssim tl_s \implies \boxed{\Gamma} \vdash hd_t; tl_t \lesssim hd_s; tl_t \tag{SEQ}$$

which splits the proof into two parts: one for the hd and the other for tl.

A drawback of this formulation is that there is no way for the subproof about hd to deliver coinductive progress to the latter subproof about tl. Any coinductive progress made in hd gets lost.

To see this more clearly, consider the following example inspired from compiler optimizations (loop rotation or polyhedral optimization) where the **skip** means an unfolding step for the while loop in underlying operational semantics.

$$L \triangleq \texttt{loop\{ skip; x; y \}; x} \sqsubseteq_{\text{beh}} \texttt{x; loop\{ skip; y; x \}} \triangleq R \tag{REORDER}$$

Our goal is to prove the above while keeping these unknown code chunks, x and y, as opaque: in other words, we want to prove it *without resorting to case-analysis* on x and y. In ISim, our best effort is as follows, with SR $\triangleq \{(L, R)\}$:

$$\begin{array}{lll}
 & \boxed{\text{SR}} \vdash \texttt{loop\{ skip; x; y \}; x} \lesssim \texttt{x; loop\{ skip; y; x \}} & \\
\iff & \boxed{\text{SR}} \vdash \texttt{skip; x; y; L} \lesssim \texttt{x; skip; y; R} & \text{by unfolding} \\
\impliedby & \boxed{\text{SR}} \vdash \texttt{x; y; L} \lesssim \texttt{x; skip; y; R} & \text{by STEP-TGT} \\
\impliedby & \boxed{\text{SR}} \vdash \texttt{y; L} \lesssim \texttt{skip; y; R} & \text{by SEQ and reflexivity} \\
\impliedby & \boxed{\text{SR}} \vdash \texttt{y; L} \lesssim \texttt{y; R} & \text{by STEP-SRC} \\
\impliedby & \boxed{\text{SR}} \vdash \texttt{L} \lesssim \texttt{R} & \text{by SEQ and reflexivity} \\
\not\impliedby & (L, R) \in \text{SR} & \text{by COIND?}
\end{array}$$

We first execute **skip** with STEP-TGT, discharge the same x on both sides with SEQ and reflexivity, execute **skip** with STEP-SRC, and discharge the same y as we did with x. Now we have reached the original state, but we cannot conclude the proof with COIND since coinductive hypotheses are still guarded! However, it is clear that—morally—there was coinductive progress since we have at least executed **skip** on both sides.

Intuitively, the problem here is as follows. We want to make coinductive progress out of **skip**s on both sides by executing them synchronously via STEP-BOTH. Unfortunately, this attempt is blocked by the opaque code chunk x—it stands against us as a 'wall' that needs to be executed synchronously (via SEQ and reflexivity). This forces us to execute **skip**s separately, wasting "partial progresses."

At the high-level, the way we address this issue in FreeSim is as follows:

While x still act as a wall in FreeSim and we execute **skip**s separately, (i) each **skip** now contributes partial progress (a dashed arrow turning into a solid arrow in the diagram), and (ii) our stronger reasoning principles allow these partial progresses to "penetrate" the wall.

Specifically, we can formulate the following stronger versions of SEQ rule and reflexivity:

$$\text{REFL+}\ \frac{\top}{\Gamma^? \vdash e\ {}_{pt}\!\overset{+}{\lesssim}_{ps}\ e\ \{pt'\ ps'.\ pt' = pt \wedge ps' = ps\}} \qquad \text{SEQ+}\ \frac{\Gamma^? \vdash hd_t\ {}_{pt}\!\overset{+}{\lesssim}_{ps}\ hd_s\ \{\Psi\} \qquad \forall (pt', ps') \in \Psi.\, \Gamma^? \vdash tl_t\ {}_{pt'}\!\overset{+}{\lesssim}_{ps'}\ tl_s}{\Gamma^? \vdash hd_t; tl_t\ {}_{pt}\!\overset{+}{\lesssim}_{ps}\ hd_s; tl_t}$$

First, to state such an additional power of transferring partial progresses from one to another (*i.e.*, penetrating the wall), we equip FreeSim with *postcondition*. Postcondition, optionally given in the parenthesis after the simulation notation, states possible values of progress indices when it returns.

With this, the strengthened reflexivity (REFL+) now additionally ensures that it preserves the given progress indices: *i.e.*, given the indices pt and ps, it returns (pt' and ps') the same indices. Then, the strengthened sequence rule (SEQ+) now additionally allows picking a "contract" Ψ between these two separate verifications: the former verification ensures that it returns indices satisfying Ψ, and the later relies upon it that it begins simulation with progress indices satisfying Ψ.

Now we revisit REORDER with FreeSim. With $\mathrm{SR} \triangleq \{(L, R, \bot, \bot)\}$, the proof is as follows:

$$
\begin{array}{lll}
 & \boxed{\mathrm{SR}} \vdash \texttt{loop}\{\ \texttt{skip};\ \mathtt{X};\ \mathtt{Y}\ \};\ \mathtt{X} \ {}_{\bot}\overset{+}{\lesssim}_{\bot}\ \mathtt{X};\ \texttt{loop}\{\ \texttt{skip};\ \mathtt{Y};\ \mathtt{X}\ \} & \\
\Longleftrightarrow & \boxed{\mathrm{SR}} \vdash \texttt{skip};\ \mathtt{X};\ \mathtt{Y};\ \mathtt{L} \ {}_{\bot}\overset{+}{\lesssim}_{\bot}\ \mathtt{X};\ \texttt{skip};\ \mathtt{Y};\ \mathtt{R} & \text{by unfolding} \\
\Longleftarrow & \boxed{\mathrm{SR}} \vdash \mathtt{X};\ \mathtt{Y};\ \mathtt{L} \ {}_{\top}\overset{+}{\lesssim}_{\bot}\ \mathtt{X};\ \texttt{skip};\ \mathtt{Y};\ \mathtt{R} & \text{by STEP-TGT} \\
\Longleftarrow & \boxed{\mathrm{SR}} \vdash \mathtt{Y};\ \mathtt{L} \ {}_{\top}\overset{+}{\lesssim}_{\bot}\ \texttt{skip};\ \mathtt{Y};\ \mathtt{R} & \text{by SEQ+ and REFL+} \\
\Longleftarrow & \boxed{\mathrm{SR}} \vdash \mathtt{Y};\ \mathtt{L} \ {}_{\top}\overset{+}{\lesssim}_{\top}\ \mathtt{Y};\ \mathtt{R} & \text{by STEP-SRC} \\
\Longleftarrow & \boxed{\mathrm{SR}} \vdash \mathtt{L} \ {}_{\top}\overset{+}{\lesssim}_{\top}\ \mathtt{R} & \text{by SEQ+ and REFL+} \\
\Longleftarrow & \boxed{\mathrm{SR}} \vdash \mathtt{L} \ {}_{\bot}\overset{+}{\lesssim}_{\bot}\ \mathtt{R} & \text{by COIND-PROG} \\
\Longleftarrow & (L, R, \bot, \bot) \in \mathrm{SR} & \text{by COIND} \quad \square
\end{array}
$$

The proof has a similar structure as before, with the difference that (i) executing **skip** now makes partial progress (setting the flag to $\top$), and (ii) that partial progress is preserved along the applications of SEQ+ and REFL+. These together constitute full coinductive progress at the end and we can now conclude the proof with coinduction.

In this subsection, we have seen how stronger reasoning principles in FreeSim allowed a new style of proof. The key difference comes from our additional ability to state partial progress. In our experience, this is in general useful that it facilitates unary (asynchronous) reasoning.

3.3 Replayability

As shown in previous sections, FreeSim sets itself apart from ESim or ISim in terms of usability. At the same time, however, it turns out that these three simulations are *propositionally* equivalent (see: §5): a statement provable in one simulation is "somehow" also provable in others. This discrepancy led us to devise a *stronger* notion than mere propositional implication, dubbed **stepwise replayability** ("replayability" for short).

Replayability. Intuitively, given two proof techniques A and B, replayability $A \blacktriangleright B$ (read as "A is replayable in B" or "B can replay A") says that B can emulate A, meaning that B can provide the same user interface as that of A. Specifically, $A \blacktriangleright B$ states that

Given the same proof state in A and B, whichever rule A executes,
B can execute a sequence of rules that results in the same proof state again.

It is clear that replayability ($A \blacktriangleright B$) implies propositional implication ($A \Longrightarrow B$): any proof in A (a sequence of rule applications) can be translated into B via the above *rule-wise* (*i.e.*, local) translations. Then, what are the key differences between these two notions?

First, in propositional implication, the proof in B can be constructed after looking at the full (*i.e.*, global) proof in A. Indeed, in our proof of implication from ISim to ESim (see §5), we look at the full (global) proof in ISim in order to set up a proper stuttering index *upfront*. To put it another way, one is allowed to construct a proof in B by *looking into the future* via the given full proof in A. Replayability removes this ability to look into the future—recall that replayability requires a rule-wise (*i.e.*, local) translation—and adequately compares the usability of A and B. Second, in propositional implication, it is okay to find a proof in B that takes a very different intermediate proof states from the given proof in A. One well-known such an example is an implication proof in CompCert [Leroy 2006] (see §6 for more). This is in contrast to replayability, where the translated proof follows exactly the same intermediate proof states as the given one.

To see this more concretely, consider the following minimal, contrived game. The game begins in $(0, 0)$ and the player wins the game if the player reaches certain states, called "winning state"s. The player can choose one of two characters, F and G, with the following actions:

$$[Fa]\ (x, y) \rightsquigarrow (x + 1, y) \qquad [Fb]\ (x, y) \rightsquigarrow (x, y + 1) \qquad [Ga]\ \forall n \in N, (x, y) \rightsquigarrow (x + n, y + 1)$$

Winning states have positive coordinates and are initially invisible to the player. When a character reaches (x, y), the player notices all winning states in $(x, \text{-})$. Now, consider a play with a character F when the winning state is in $(2, 1)$.

$(0, 0)$ // Realizes there is no winning state in $(0, \text{-})$.
$\rightsquigarrow_{Fa}\ (1, 0)$ // Realizes there is no winning state in $(1, \text{-})$.
$\rightsquigarrow_{Fa}\ (2, 0)$ // Realizes there is a winning state in $(2, 1)$.
$\rightsquigarrow_{Fb}\ (2, 1)$ // Wins the game.

However, if one plays with the character G, it gets tricky.

$(0, 0)$ // Realizes there is no winning state in $(0, \text{-})$.
// Unclear which n to use for the next step, just pick 1.
$\rightsquigarrow_{Ga}\ (1, 1)$ // Realizes there is no winning state in $(1, \text{-})$.
// Unclear which n to use for the next step, just pick 1.
$\rightsquigarrow_{Ga}\ (2, 2)$ // Realizes there is a winning state in $(2, 1)$.
// However, $(2, 1)$ already became unreachable. Need to restart the game.

Without knowing $(2, 1)$ is a winning state in advance, the player easily ends up in an unprovable goal, and the player needs to restart the game (backtracking). Note, however, that any game that is winnable with F is also winnable with G (though it may require few restarts).

This example captures the difference between propositional implication and replayability. Here, F and G are propositionally equivalent ($F \Longleftrightarrow G$). Yet, the notion of replayability is sensitive enough to distinguish F and G. Check that G ▬▬▸ F holds while F ▬▬▸ G does not hold (the action $[Fa]$ does not have a matching action in G). This is in contrast to propositional implication, where $F \Longrightarrow G$ holds. Such an implication proof indeed (i) looks the given full play in F (to figure out the winning states in advance), and (ii) takes a different path from the given play in F.

In the rest of this section, we aim to show ESim ▬▬▸ FreeSim and ISim ▬▬▸ FreeSim; the reverse does not hold since there is no matching rule for COIND-PROG. In Fig. 5, we present how a selected rules of ESim and ISim can be translated into a sequence of rules of the FreeSim.

Replaying ESim. Replayability proof very much resembles simulation proof, in the sense that it 'simulates' simulations: (i) we define an embedding from the original proof state into the replayer's proof state (just like defining SR), and (ii) prove that for any rule application in the original simulation, there exists a matching rule application(s) in the replayer that results in a proof state satisfying (i) again.

Thus, we start by defining such an embedding. Given a well-founded order $\mathbb{J}$ for ESim, we use $\mathbb{I} \triangleq \{\top\} \uplus \mathbb{J}$ and $o_0 <_{\mathbb{I}} o_1 \triangleq (o_0 \neq \top \wedge o_1 = \top) \vee (o_0 <_{\mathbb{J}} o_1)$ for FreeSim. Then, we use the following:

$$\Gamma^? \vdash s \mathrel{\overline{\lesssim}_i} t \qquad \longrightarrow \qquad \{(s, t, i, i) \mid (s, t, i) \in \Gamma\}^? \vdash s \mathrel{{}_{\mathbb{I}}\overset{+}{\lesssim}_{\mathbb{I}}} t \qquad \text{(EMBEDDING)}$$

That is, we pick both progress indices to be equal to the stuttering index.

Now we explain how to replay a few selected rules. For the STEP-TGT rule of ESim, we apply STEP-TGT, followed by COIND-PROG: since our progress indices match the stuttering index, COIND-PROG allows us to unguard the coinductive hypothesis by decrementing both indices to i'. The resulting proof state is again an embedding (EMBEDDING) of the original proof state.

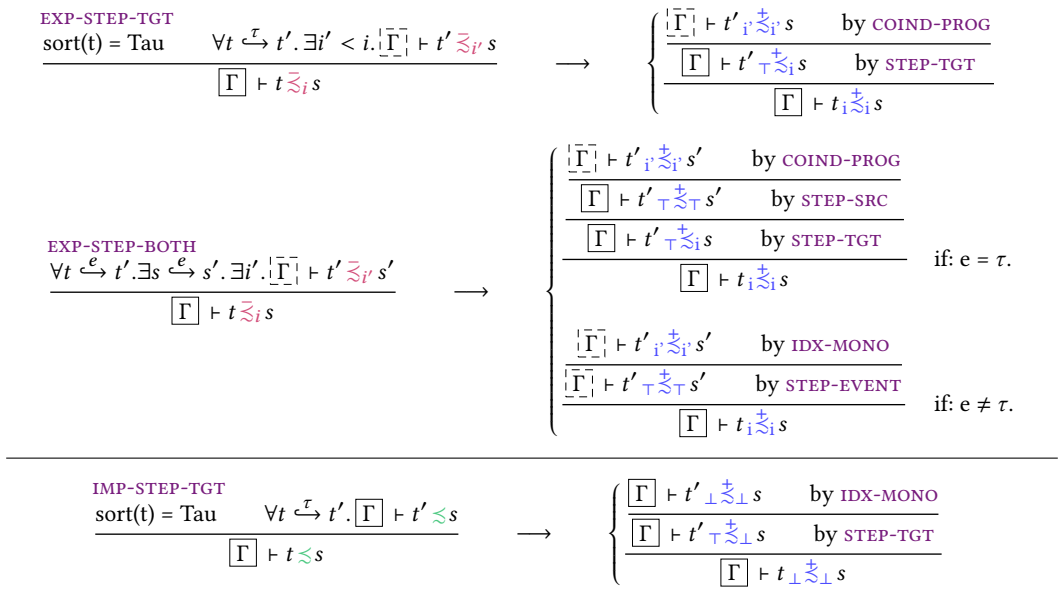


Fig. 5. Replaying previous simulations in freely-stuttering simulation.

Similarly, we replay STEP-BOTH of ESim as follows. We start with a case analysis on the event e emitted in source and target. If it is a silent event τ, we asynchronously step the target and the source with STEP-SRC and STEP-TGT, respectively, and apply COIND-PROG to unguard the coinductive hypothesis. Otherwise, STEP-EVENT allows us to step synchronously for a visible event.

Replaying ISim. When replaying ISim, we simply pick a well-founded order $\mathbb{I}$ consisting of two comparable elements: $\bot <_{\mathbb{I}} \top$. Then, we use the following embedding which puts $\bot$to both indices:

$$\boxed{\Gamma} \vdash s \precsim t \quad \longrightarrow \quad \boxed{\{(s,t,\bot,\bot) \mid (s,t) \in \Gamma\}} \vdash s \; {}_{\bot}\overset{+}{\precsim}_{\bot} \; t \qquad \text{(EMBEDDING)}$$

For the STEP-TGT rule of ISim, we apply STEP-TGT, followed by IDX-MONO: since the original rule does not unguard coinductive hypothesis, we do not need to (and cannot) apply COIND-PROG. Instead, we apply IDX-MONO to "forget" the partial progress we have made, which results in the proof state again an embedding (EMBEDDING) of the original proof state.

The fact that FreeSim can replay both styles not only shows that FreeSim subsumes both styles, but also suggests that one can "mix" these two styles *in a single proof* (see: §5.2 and §6.1).

3.4 Adding New Primitives: A Case With Dual Non-Determinism

Finally, we consider how FreeSim scales better than ESim and ISim when adding new primitives to the notion of STS. Recall that FreeSim has *decoupled* the program execution and coinductive progress, making STEP-BOTH obsolete. Now, suppose we add a new Sort, say X, to the notion of STS. In FreeSim, we just need to add two rules for asynchronously executing X in target and source. However, in ISim and ESim, one needs to further add rules to synchronously execute (i) both X in target and source, (ii) X in target and Tau in source, and (iii) vice versa. These synchronous rules are largely a duplication of asynchronous rules and make the definition unnecessarily complex.

To make this argument concrete, we extend our STS (and FreeSim, correspondingly) to handle *dual non-determinism* which is recently being rediscovered and finding interesting uses [Koenig and Shao 2020; Sammler et al. 2023; Song et al. 2023]. Specifically, we extend the Sort component

RULES FOR PROGRAM EXECUTION

$$\frac{\text{sort(t)} = \text{ATau} \qquad \exists t \overset{\tau}{\hookrightarrow} t'.\, \Gamma^? \vdash t' \; {}_{\top}\overset{+}{\lesssim}_{ps} \; s}{\Gamma^? \vdash t \; {}_{pt}\overset{+}{\lesssim}_{ps} \; s}\ \text{STEP-TGT-ANG} \qquad \frac{\text{sort(s)} = \text{ATau} \qquad \forall s \overset{\tau}{\hookrightarrow} s'.\, \Gamma^? \vdash t \; {}_{pt}\overset{+}{\lesssim}_{\top} \; s'}{\Gamma^? \vdash t \; {}_{pt}\overset{+}{\lesssim}_{ps} \; s}\ \text{STEP-SRC-ANG}$$

Fig. 6. Freely-stuttering simulation extended for dual non-determinism.

of our STS as follows:

$$\text{Sort} \triangleq \{\text{DTau}\} \uplus \{\text{ATau}\} \uplus \{\text{Vis}\} \uplus \{\text{Ret}\}$$

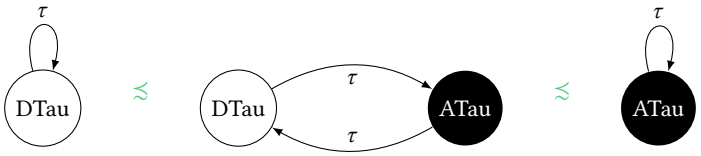
To highlight the symmetric nature, we rename Tau into DTau (for "demonic") and add a new Sort, ATau (for "angelic"). Similarly to DTau, states of sort ATau can only take silent steps.

ATau is interpreted as the dual of DTau: the behavior of a state in DTau is a union of the behaviors of successor states, and for ATau it is an *intersection* (the dual of union). Correspondingly, the rules for ATau (shown in Fig. 6) are exactly the dual of the rules for DTau. That is, STEP-TGT-ANG performs an angelic step in the target and is dual to STEP-TGT: the universal quantifier has been swapped to an existential quantifier (and similarly for STEP-SRC-ANG).

Adding these two rules is all we need to fully support dual non-determinism. As we will see in §6, this simple definition satisfies all the meta-theoretic properties one would expect. Actually, the simplicity of this definition is the key to proving non-trivial properties such as transitivity.

Comparison with ESim/ISim. For comparison, let us consider adding angelic non-determinism to ISim. As mentioned above, we need two rules for asynchronous execution and three additional rules for synchronous execution. That is, we end up with *four* rules for synchronous execution: we have two cases (DTau or ATau in the target) multiplied by two cases (DTau or ATau in the source).

An astute reader might ask: why do we care about supporting all four synchronous rules? It is clear that *symmetric* synchronous rules (one for executing DTau on both sides and one for ATau on both sides) are essential for the **reflexivity** to hold. Then, *asymmetric* synchronous rules (the other two) are essential for the **transitivity** to hold. To see this, consider the following example:



These three STS all represent silent divergence. Suppose that transitivity holds but we do not have asymmetric rules. Still, the above two stepwise simulations are easily provable (using symmetric rules and asynchronous rules). Then, by transitive composition of step-wise simulations, an end-to-end simulation between the left and the right STS holds. However, since we do not have asymmetric rules, the end-to-end simulation cannot actually be proven, which leads to a contradiction.

Indeed, our best efforts to define ESim/ISim supporting dual non-determinism have resulted in a definition that has all four cases. Note that we had to work on these definitions when proving the aforementioned equivalence results, presented in §5.

In a more general sense, having synchronous rules (as in ESim/ISim) is less ideal for the following reasons. First, it leads to a combinatorial blow-up when adding a new primitive, and this could be a practical problem when proving properties that require case-analysis on the definition of the simulation (*e.g.*, transitivity). Second, there will be a lot of duplication between synchronous rules and asynchronous rules. Third, it is *not* modular: adding a new primitive requires considering its interaction with existing primitives (those *asymmetric* rules). FreeSim is free from these problems.

$$
\begin{array}{l}
\textbf{FreeSimF}(\Psi \in \mathcal{P}(\mathbb{I}\times\mathbb{I}\times \mathrm{Val}\times\mathrm{Val}))(C \in \mathcal{P}(X_{\mathbb{T}}\times X_{\mathbb{S}}\times\mathbb{I}\times\mathbb{I})) \overset{\text{ind}}{=} \{(t,s,pt,ps) \mid \\
\quad (\mathrm{sort(t)} = \mathrm{DTau} \wedge \forall\, t \xrightarrow{\tau} t'.\ (t',s,\top,ps) \in \textbf{FreeSimF}(\Psi)(C)) && \text{// (1) STEP-TGT} \\
\vee\ (\mathrm{sort(t)} = \mathrm{ATau} \wedge \exists\, t \xrightarrow{\tau} t'.\ (t',s,\top,ps) \in \textbf{FreeSimF}(\Psi)(C)) && \text{// (2) STEP-TGT-ANG} \\
\vee\ (\mathrm{sort(s)} = \mathrm{DTau} \wedge \exists\, s \xrightarrow{\tau} s'.\ (t,s',pt,\top) \in \textbf{FreeSimF}(\Psi)(C)) && \text{// (3) STEP-SRC} \\
\vee\ (\mathrm{sort(s)} = \mathrm{ATau} \wedge \forall\, s \xrightarrow{\tau} s'.\ (t,s',pt,\top) \in \textbf{FreeSimF}(\Psi)(C)) && \text{// (4) STEP-SRC-ANG} \\
\vee\ (\mathrm{sort(t)} = \mathrm{sort(s)} = \mathrm{Vis} \wedge \forall\, t \xrightarrow{e} t'. \exists\, s \xrightarrow{e} s'.\ (t',s',\top,\top) \in \textbf{FreeSimF}(\Psi)(C)) && \text{// (5) STEP-EVENT} \\
\vee\ (\exists\, \mathrm{r_t}\ \mathrm{r_s}\ (pt' \le pt)\ (ps' \le ps).\ \mathrm{sort(t)} = \mathrm{Ret(r_t)} \wedge \mathrm{sort(s)} = \mathrm{Ret(r_s)} \wedge (pt', ps', \mathrm{r_t}, \mathrm{r_s}) \in \Psi) && \text{// (6) RETURN} \\
\vee\ (\exists\, (pt' < pt)\ (ps' < ps).\ (t,s,pt',ps') \in C)\} && \text{// (7) COIND-PROG}
\end{array}
$$

$$
t\ {}_{pt}^{+}\!\gtrsim_{ps}\ s\ \{\Psi\} \triangleq (t,s,pt,ps) \in \nu\, \textbf{FreeSimF}(\Psi) \qquad t\ {}_{pt}^{+}\!\gtrsim_{ps}\ s \triangleq t\ {}_{pt}^{+}\!\gtrsim_{ps}\ s\ \{\lambda\ \_\ \_\ \mathrm{r_t}\ \mathrm{r_s}.\ \mathrm{r_t} = \mathrm{r_s}\}
$$

Fig. 7. Definitions of FreeSim.

4 FORMALIZATION AND KEY LEMMAS

In this section, we present the formal definition of FreeSim and the proof system for it.

4.1 Definition of FreeSim

As mentioned, FreeSim is defined in a coinductive manner, which formally means that it is defined as a *greatest fixed point* of some functor. In general, for a given set C and a monotone functor $F \in \mathcal{P}(C) \to \mathcal{P}(C)$, its greatest fixed point $\nu F \in \mathcal{P}(C)$ is defined as the largest set satisfying the equation $\nu F = F(\nu F)$. Given a postcondition Ψ, FreeSim can be defined as the greatest fixed point of $\textbf{FreeSimF}(\Psi)$ (given in Fig. 7) which is a functor over a powerset of a quadruple comprising target state, source state, and progress indices. We omit Ψ when it is simply equality.

One can see that each case of **FreeSimF** corresponds to the rules in Fig. 4 and Fig. 6. Recall that the only rule that unguards coinductive hypothesis is COIND-PROG. Such a fact is reflected in the above definition that the corresponding case (7) is making a *corecursion* to C, while all other cases are making a *recursion* to **FreeSimF**.

4.2 Proof System

Admittedly, however, the above definition looks quite different from the interface we presented throughout the paper. Specifically, there are two questions remaining to be answered: (i) how are the guarded/unguarded hypotheses formalized, and (ii) where is the rule IDX-MONO, which is missing in Fig. 7? In this subsection, we aim to answer these questions, by making use of features commonly provided by modern coinduction libraries [Hur et al. 2013; Pous 2016; Zakowski et al. 2020]: *parameterized coinduction* and *up-to* techniques.

Among these libraries, we pick GPaco [Zakowski et al. 2020] (an extension of Paco [Hur et al. 2013]) as the basis of our presentation: our Coq development is based on GPaco and we would like to keep the presentation close.[2] The core proof system of GPaco is presented in Fig. 8.[3] Note that the `coinduction` library [Pous 2016] also provides a similar proof system.

Intuitively, the proof system of GPaco could be understood by contrasting it with the proof system for strong induction. Strong induction allows one to conclude the proof when the argument has *decreased*. Similarly, in coinduction proofs, one can conclude the proof with coinductive hypotheses when *coinductive progress* has been made. GPaco comes with the notion of guarded/unguarded coinductive hypotheses to represent the coinductive hypotheses before/after coinductive progress. Specifically, this is achieved by generalizing the vanilla greatest fixed point (νF) into a *parameterized*

[2] Paco is a shorthand for **pa**rameterized **co**induction, and GPaco for **g**eneralized **pa**rameterized **co**induction.

[3] Both our Coq development and the original GPaco proof system allows *incremental reasoning* [Hur et al. 2013], meaning that it can add more guarded coinductive hypotheses during the proof. However, this requires adding an additional parameter in our notation as follows: $[\boxed{\Gamma_0}\,[\boxed{\Gamma_1}\,] \vdash \nu F]$. Thus, we omit it in our presentation for uniformity and simplicity.

$C \in \mathbf{Set} \qquad F, G \in \mathcal{P}(C) \to \mathcal{P}(C) \qquad \Gamma, \nu F, \Gamma^? \vDash \nu F \in \mathcal{P}(C)$

Rules in GPaco proof system

$$\textsc{gpaco-init}\ \frac{\Gamma \subseteq (\boxed{\Gamma} \vDash \nu F)}{\Gamma \subseteq \nu F} \qquad \textsc{gpaco-step}\ \frac{\top}{F\,(\overline{\boxed{\Gamma}} \vDash \nu F) \subseteq \Gamma^? \vDash \nu F} \qquad \textsc{gpaco-coind}\ \frac{\top}{\Gamma \subseteq (\overline{\boxed{\Gamma}} \vDash \nu F)} \qquad \textsc{gpaco-upto}\ \frac{G \text{ is compatible with F}}{G\,(\Gamma^? \vDash \nu F) \subseteq \Gamma^? \vDash \nu F}$$

Fig. 8. GPaco proof system.

greatest fixed point which parameterizes over coinductive hypotheses that could be either guarded ($\boxed{\Gamma} \vDash \nu F$) or unguarded ($\overline{\boxed{\Gamma}} \vDash \nu F$).

Now we look at the rules in Fig. 8. In order to establish that Γ is contained in νF, one can initialize the coinduction proof by putting Γ into the coinductive hypothesis (GPACO-INIT). The coinductive hypothesis is initially guarded, and so can not be used directly to complete the proof. After coinductive progress has been made by GPACO-STEP, the hypothesis becomes unguarded. Finally, GPACO-COIND can complete the proof using the unguarded hypothesis.

As an example, consider Tarski's theorem, a basic reasoning principle for the vanilla greatest fixed point:

$$\Gamma \subseteq F\,\Gamma \quad \Longrightarrow \quad \Gamma \subseteq \nu F \qquad \text{(TARSKI)}$$

Such a reasoning principle could be replayed in GPaco as follows. By applying GPACO-INIT and GPACO-STEP in the goal, it remains to prove: $\Gamma \subseteq F\,(\overline{\boxed{\Gamma}} \vDash \nu F)$. By applying GPACO-COIND with monotonicity of F, it suffices to prove $\Gamma \subseteq F\,\Gamma$, which is the premise of TARSKI.

Now, let us get back to our first question. With GPaco, we define FreeSim equipped with coinductive hypotheses as follows:

$$(\overline{\boxed{\Gamma}} \vDash t \overset{+}{\lesssim}_{pt,ps} s\ \{\Psi\}) \triangleq (\overline{\boxed{\Gamma}} \vDash \nu\ \mathbf{FreeSimF}(\Psi))(t, s, pt, ps)$$
$$(\boxed{\Gamma} \vDash t \overset{+}{\lesssim}_{pt,ps} s\ \{\Psi\}) \triangleq (\boxed{\Gamma} \vDash \nu\ \mathbf{FreeSimF}(\Psi))(t, s, pt, ps)$$

Now, it is easy to see the connection between the rules in Fig. 4 and the rules in Fig. 8. First, if we think of ADEQUACY in two steps:

$$\begin{cases} (\forall (t, s, pt, ps) \in \mathrm{SR}.\, \boxed{\mathrm{SR}} \vDash t \overset{+}{\lesssim}_{pt,ps} s) \implies (\forall (t, s, pt, ps) \in \mathrm{SR}.\, t \overset{+}{\lesssim}_{pt,ps} s) \\ (t \overset{+}{\lesssim}_{pt,ps} s) \implies (\mathrm{Beh}(t) \subseteq \mathrm{Beh}(s)) \end{cases}$$

The former directly corresponds to GPACO-INIT. The latter will be discussed in Theorem 4.1. Similarly, COIND is realized with the rule GPACO-COIND. Finally, COIND-PROG, together with program execution rules (which are already reflected in the definition of **FreeSimF**), is realized with GPACO-STEP.

Now, for the second question: IDX-MONO is derived using GPACO-UPTO. Up-to techniques [Milner 1989; Pous 2016; Pous and Sangiorgi 2011; Sangiorgi 1998] make coinduction proofs easier by extending (parameterized) greatest fixed point in the goal by putting a *compatible* up-to functor G. GPaco has native support for up-to techniques, and we use it to derive IDX-MONO (and much more in §6.2). Since GPACO-UPTO does not unguard hypotheses, the same goes for IDX-MONO.

4.3 Behavior

Now we define the notion of *behavior* for STS, and then discuss the adequacy of FreeSim. Our definition of behavior, given in Fig. 9, follows literature [Gäher et al. 2022; Leroy 2006; Song et al. 2023; Zhao et al. 2012] and there is nothing novel here. Thus, we keep the discussion short.

The behavior of a program state, Beh, denotes the set of possible *traces* the state can invoke when executed. Trace is a set of traces, where a trace records visible events of type $\mathcal{E}$ and can (i) have infinitely many events, (ii) terminate with a return value (Term), or (iii) silently diverge (Diverge).

The definition of Beh is a mixture of recursion and corecursion. That is, Beh is defined as a greatest fixpoint of BehF (like FreeSim is defined with **FreeSimF**) where BehF itself is inductively

$$\mathrm{Trace} \overset{\mathrm{coind}}{=} \{e :: tr \mid e \in \mathcal{E}, tr \in \mathrm{Trace}\} \uplus \{\mathrm{Term}\ v \mid v \in \mathrm{Val}\} \uplus \{\mathrm{Diverge}\}$$

$$\mathrm{Beh}F(C \in X \to \mathbb{P}(\mathrm{Trace})) \in X \to \mathbb{P}(\mathrm{Trace}) \overset{\mathrm{ind}}{=} \lambda s.\ \{\mathrm{Diverge}\}|_{s \in \mathrm{div}} \cup (\bigcup_{\{s' \mid s \xrightarrow{\tau} s'\}} \mathrm{Beh}F(s'))|_{\mathrm{sort}(s)=\mathrm{DTau}} \cup (\bigcap_{\{s' \mid s \xrightarrow{\tau} s'\}} \mathrm{Beh}F(s'))|_{\mathrm{sort}(s)=\mathrm{ATau}} \cup (\bigcup_{\{(e,s') \mid s \xrightarrow{e} s'\}} e :: C(s'))|_{\mathrm{sort}(s)=\mathrm{Vis}} \cup \{\mathrm{Term}\ v \mid \mathrm{sort}(s) = \mathrm{Ret}(v)\}$$

$$\mathrm{Beh} \triangleq \nu\, \mathrm{Beh}F$$

$$\mathrm{div} \in \mathbb{P}(X) \overset{\mathrm{coind}}{=} \{s \mid (\mathrm{sort}(s) = \mathrm{DTau} \wedge \exists s'.s \xrightarrow{\tau} s' \wedge s' \in \mathrm{div}) \vee (\mathrm{sort}(s) = \mathrm{ATau} \wedge \forall s'.s \xrightarrow{\tau} s' \to s' \in \mathrm{div})\}$$

Fig. 9. Definitions of trace and behavior.

defined (like **FreeSimF**). In the definition of Beh$F$, the recursive occurrences via Beh$F$ are (inductive) recursions, and the recursive occurrence via C is a (coinductive) corecursion. Then the behavior Beh(s) of a state s contains every possible trace of s. If the state s silently diverges, judged by div predicate, it emits the trace Diverge. If sort(s) is DTau/ATau, it takes the union/intersection of every possible successive behavior of s. If sort(s) is Vis, it appends the visible event e to each possible successive behavior of s. If sort(s) is Ret, it emits the trace Term.

4.4 Adequacy

Now we state the adequacy theorem which says that FreeSim implies behavioral refinement.

Theorem 4.1 (Adequacy). *For every t, s, pt, and ps, $t \ {}_{pt}\overset{+}{\lesssim}_{ps} s \Rightarrow Beh(t) \subseteq Beh(s)$.*

Proof. For the sake of space, we outline the proof for terminating and silently diverging cases: it can be easily extended to handle visible events. That is, we prove the following two propositions:

$$(i)\ \forall\, t\ s\ pt\ ps.\ t \ {}_{pt}\overset{+}{\lesssim}_{ps} s \Rightarrow (\forall\, \mathrm{r}.\, \mathrm{Term\ r} \in \mathrm{Beh}(t) \Rightarrow \mathrm{Term\ r} \in \mathrm{Beh}(s))$$
$$(ii)\ \forall\, t\ s\ pt\ ps.\ t \ {}_{pt}\overset{+}{\lesssim}_{ps} s \Rightarrow (\mathrm{Diverge} \in \mathrm{Beh}(t) \Rightarrow \mathrm{Diverge} \in \mathrm{Beh}(s))$$

Proof outline of (i): we use triple nested inductions. **(A)** Target terminates within finite number of steps, and we use induction on the remaining number of steps. **(B)** We use induction on *target* progress index.[4] **(C)** We use structural induction on the simulation. The structural induction gives us three cases, all of which are directly discharged by induction hypotheses: (1) when target takes a step, it gets discharged with **(A)**, (2) when progress indices get decremented (making coinductive progress), it gets discharged with **(B)**, and (3) when source takes a step, it gets discharged with **(C)**.

Proof outline of (ii): we use one coinduction and two inductions. **(A)** Source diverges, and we use coinduction on div. **(B)** We use induction on *source* progress index. **(C)** We use structural induction on the simulation. The structural induction gives us three cases, all of which are directly discharged by (co)induction hypotheses: (1) when source takes a step, it gets discharged with **(A)**. (2) when progress indices get decremented (making coinductive progress), it gets discharged with **(B)**, and (3) when target takes a step, it gets discharged with **(C)**.

□

5 COMPARISON BETWEEN DIFFERENT SIMULATIONS

Now we investigate the relationship between FreeSim and other simulations (ISim and ESim).

5.1 Definition of ISim

Similar to FreeSim, ISim is defined as a greatest fixed point induced by a functor **ISimF** (Fig. 10).

Here we consider the full definition considering dual non-determinism for comparison with FreeSim. A definition considering only the demonic non-determinism (corresponding to the interface given in Fig. 3) can be simply obtained by ignoring all the cases with ATau.

There are two things to note in this definition. First, recall how ISim ensured finiteness of stuttering: it did *not* unguard coinductive hypotheses in *asynchronous* steps. That can be checked in

[4] Recall that progress indices are well-founded; we use standard well-founded induction.

$$\begin{aligned}
&\mathbf{ISimF}(\Psi \in \mathcal{P}(\mathrm{Val}\times\mathrm{Val}))(C \in \mathcal{P}(X_{\mathbb{T}}\times X_{\mathbb{S}})) \overset{\text{ind}}{=} \{(t,s)\mid\\
&\quad(\mathrm{sort(t)} = \mathrm{DTau} \wedge \forall\, t \overset{\tau}{\hookrightarrow} t'.\\
&\qquad((t',s) \in \mathbf{ISimF}(\Psi)(C) \vee (\mathrm{sort(s)} = \mathrm{DTau} \wedge \exists\, s \overset{\tau}{\hookrightarrow} s'.\,(t',s') \in C) \vee (\mathrm{sort(s)} = \mathrm{ATau} \wedge \forall\, s \overset{\tau}{\hookrightarrow} s'.\,(t',s') \in C)))\\
&\vee (\mathrm{sort(s)} = \mathrm{ATau} \wedge \forall\, s \overset{\tau}{\hookrightarrow} s'.\\
&\qquad((t,s') \in \mathbf{ISimF}(\Psi)(C) \vee (\mathrm{sort(t)} = \mathrm{DTau} \wedge \forall\, t \overset{\tau}{\hookrightarrow} t'.\,(t',s') \in C) \vee (\mathrm{sort(t)} = \mathrm{ATau} \wedge \exists\, t \overset{\tau}{\hookrightarrow} t'.\,(t',s') \in C)))\\
&\vee (\mathrm{sort(t)} = \mathrm{ATau} \wedge \exists\, t \overset{\tau}{\hookrightarrow} t'.\\
&\qquad((t',s) \in \mathbf{ISimF}(\Psi)(C) \vee (\mathrm{sort(s)} = \mathrm{DTau} \wedge \exists\, s \overset{\tau}{\hookrightarrow} s'.\,(t',s') \in C) \vee (\mathrm{sort(s)} = \mathrm{ATau} \wedge \forall\, s \overset{\tau}{\hookrightarrow} s'.\,(t',s') \in C)))\\
&\vee (\mathrm{sort(s)} = \mathrm{DTau} \wedge \exists\, s \overset{\tau}{\hookrightarrow} s'.\\
&\qquad((t,s') \in \mathbf{ISimF}(\Psi)(C) \vee (\mathrm{sort(t)} = \mathrm{DTau} \wedge \forall\, t \overset{\tau}{\hookrightarrow} t'.\,(t',s') \in C) \vee (\mathrm{sort(t)} = \mathrm{ATau} \wedge \exists\, t \overset{\tau}{\hookrightarrow} t'.\,(t',s') \in C)))\\
&\vee (\mathrm{sort(t)} = \mathrm{sort(s)} = \mathrm{Vis} \wedge \forall\, t \overset{e}{\hookrightarrow} t'.\exists\, s \overset{e}{\hookrightarrow} s'.\,(t',s') \in C)\\
&\vee (\exists\, \mathrm{r_t}\ \mathrm{r_s}.\,\mathrm{sort(t)} = \mathrm{Ret(r_t)} \wedge \mathrm{sort(s)} = \mathrm{Ret(r_s)} \wedge (\mathrm{r_t},\mathrm{r_s}) \in \Psi)\}
\end{aligned}$$

$t \precsim s\ \{\Phi\} \triangleq (t,s) \in \nu\,\mathbf{ISimF}(\Psi)$ Gray-colored parts are duplication and can be safely removed (see text).

Fig. 10. Definitions of ISim.

RPL-REFL
$$\frac{\top}{F \overset{(\mathrm{id},\mathrm{id})}{\longrightarrow} F}$$

RPL-TRANS
$$\frac{F \overset{\alpha_0}{\longrightarrow} G \quad G \overset{\alpha_1}{\longrightarrow} H}{F \overset{\alpha_1\circ\alpha_0}{\longrightarrow} H}$$

RPL-COIND-STEP
$$\frac{F \overset{(\alpha,\gamma)}{\longrightarrow} G}{(\alpha \circ F \circ \gamma)[\Gamma] \vdash \nu G] \subseteq [\Gamma^? \vdash \nu G]}$$

RPL-ADEQUACY
$$\frac{F \overset{(\alpha,\gamma)}{\longrightarrow} G}{\nu F \subseteq \gamma(\nu G)}$$

Fig. 11. Properties of replayability.

the above definition that all the asynchronous cases are making *recursion* (**ISimF**), not *corecursion* (C): only the synchronous cases are making corecursion. Second, recall our claim in §3.4 about the combinatorial blow-up in adding new primitives. The definition indeed contains four cases for asynchronous executions and four cases for synchronous (both symmetric and asymmetric) executions. Furthermore, it contains few duplicated (colored gray) cases to keep the definition symmetric. Such a definition is a result of our best effort to simplify the proof (Theorem 5.4).

Similarly, we can define a functor $\mathbf{ESimF}(\Psi \in \mathcal{P}(\mathbb{I}\times\mathrm{Val}\times\mathrm{Val}))$ whose type is $(\mathcal{P}(X_{\mathbb{T}}\times X_{\mathbb{S}}\times\mathbb{I}) \to \mathcal{P}(X_{\mathbb{T}}\times X_{\mathbb{S}}\times\mathbb{I}))$, and derive a simulation with explicit stuttering as $t \precsim_i s\ \{\Psi\} \triangleq (t,s,i) \in \nu\,\mathbf{ESimF}(\Psi)$. The definition of **ESimF** is the same with **ISimF** except for stuttering index, and we omit it for space.

5.2 Replayability

In this section, we formally define replayability and use it to compare FreeSim with ESim and ISim.

Replayability is a relation between two functors (which we use to define greatest fixed points). Suppose there are two monotone functors $F \in \mathcal{P}(A) \to \mathcal{P}(A)$ and $G \in \mathcal{P}(B) \to \mathcal{P}(B)$. Since their base set cannot be compared directly ($\mathcal{P}(A)$ and $\mathcal{P}(B)$), we use *Galois connection* to lift $\mathcal{P}(A)$ to $\mathcal{P}(B)$ and to compare F and G. A Galois connection consists of a pair of monotone functions $(\alpha \in \mathcal{P}(A) \to \mathcal{P}(B), \gamma \in \mathcal{P}(B) \to \mathcal{P}(A))$ satisfying $\alpha \circ \gamma \subseteq \mathrm{id} \wedge \mathrm{id} \subseteq \gamma \circ \alpha$. As γ is uniquely determined by α, we omit γ unless necessary. Then, we define replayability with respect to a Galois connection as follows.

Definition 5.1 (Replayability). A functor $F \in \mathcal{P}(A) \to \mathcal{P}(A)$ is replayable by a functor $G \in \mathcal{P}(B) \to \mathcal{P}(B)$ with respect to (α,γ), denoted by $F \overset{(\alpha,\gamma)}{\longrightarrow} G$, if the following holds: $\alpha \circ F \subseteq G \circ \alpha$.

By the property of Galois connection, the above condition is equivalent to: $F \subseteq \gamma \circ G \circ \alpha$, $\alpha \circ F \circ \gamma \subseteq G$, and $F \circ \gamma \subseteq \gamma \circ G$. Moreover, the above condition is also equivalent to the following:

$$\forall a \in \mathcal{P}(A),\ b \in \mathcal{P}(B),\ a \sqsubseteq b \Longrightarrow Fa \sqsubseteq Gb$$

where $a \sqsubseteq b \triangleq \alpha a \subseteq b$. This formulation connects more clearly to the explanation we gave in §3.3: given the related proof state ($a \sqsubseteq b$), executing F and G on each side results in the related proof state again ($Fa \sqsubseteq Gb$).

As can be seen in Fig. 11, replayability is reflexive and transitively composable. RPL-ADEQUACY says a *whole proof* in the replayed functor F (*e.g.*, $x \subseteq vF$) could be transferred to the host functor G. In other words, replayability implies implication. RPL-COIND-STEP says something stronger: it generalizes GPACO-STEP to use any replayable functor, not just the host functor. That is, the host functor can dynamically (in the middle of the coinductive proof) employ any replayable functor only for a *single step* (not the whole proof). Therefore, multiple different replayable functors and the host functor can be freely mixed in a single proof. Such a flexibility plays a key role in §6.1.

Now, we can formalize the replayability of ESim and ISim in Fig. 5 as:

$$\mathbf{ESimF}((\lambda\ \_.\ \Psi)) \xrightarrow{\alpha} \mathbf{FreeSimF}(\lambda\ \_\ \_.\ \Psi) \text{ with } \alpha(C) = \{(t,s,pt,ps) | \exists i.\ (t,s,i) \in C \wedge i \le pt \wedge i \le ps)\}$$
$$\mathbf{ISimF}(\Psi) \xrightarrow{\alpha} \mathbf{FreeSimF}(\lambda\ \_\ \_.\ \Psi) \text{ with } \alpha(C) = \{(t,s,pt,ps) | (t,s) \in C \wedge pt' \le pt \wedge ps' \le ps)\}$$

Note that we restrict postcondition Ψ to ignore indices. Replayability of ISim is proven for any given constant index pt' and ps' smaller than $\top$. By RPL-ADEQUACY, we get the following corollaries:

COROLLARY 5.2 (ESIM IMPLIES FREESIM). *For any t, s, and index i,* $t \precsim_i s\ \{\lambda\ \_.\ \Psi\} \Rightarrow t \overset{+}{\precsim}_i s\ \{\lambda\ \_.\ \Psi\}$.

COROLLARY 5.3 (ISIM IMPLIES FREESIM). *For any t, s, and* $(pt < \top)$, $(ps < \top)$, $t \precsim s\ \{\Psi\} \Rightarrow t \overset{+}{\precsim}_{pt,ps} s\ \{\lambda\ \_.\ \Psi\}$.

5.3 Equivalence Between Simulations

As have hinted few times, despite clear differences in usability, FreeSim is *propositionally* equivalent to ESim and ISim. We use a classical trick to prove this equivalence, namely showing that (i) ESim implies ISim, (ii) ISim implies FreeSim, and (iii) FreeSim implies ESim. That is, we show:

$$\begin{array}{lrcl}
\text{(i)} & (\exists\ i.\ t \precsim_i s\ \{\lambda\ _.\ \Psi\}) & \Longrightarrow & (t \precsim s\ \{\Psi\}) \\
\text{(ii)} & (t \precsim s\ \{\Psi\}) & \Longrightarrow & (\forall\ (pt < \top)\ (ps < \top).\ t \overset{+}{\precsim}_{pt,ps} s\ \{\lambda\ _\ _.\ \Psi\}) \\
\text{(iii)} & (\exists\ pt\ ps.\ t \overset{+}{\precsim}_{pt,ps} s\ \{\lambda\ _\ _.\ \Psi\}) & \Longrightarrow & (\exists\ i.\ t \precsim_i s\ \{\lambda\ _.\ \Psi\})
\end{array}$$

One can prove (i) by induction on the index of ESim (together with coinduction); observe that ESim and ISim have the same interface, only differing by how they enforce stuttering. ESim decreases the index with stuttering, enabling the induction hypothesis to conclude the proof, and when ESim makes coinductive progress, ISim can make the same progress to conclude the proof by coinduction. Note that this proof is not replayable.[5] (ii) is already proven in Corollary 5.3. Proving (iii) is the most interesting part, where we had to construct the maximum stuttering possible by FreeSim and establish ESim with it. Interested readers can refer to the supplementary material [Cho et al. 2023].

All in all, we obtain the following:

THEOREM 5.4 (SIMULATION EQUIVALENCE). *The three simulations are propositionally equivalent:*
$\forall\ t\ s\ \Psi.\ (\exists\ pt\ ps.\ t \overset{+}{\precsim}_{pt,ps} s\{\lambda\ \_\ \_.\ \Psi\}) \Leftrightarrow (\exists\ i.\ t \precsim_i s\{\lambda\ \_.\ \Psi\}) \Leftrightarrow (t \precsim s\{\Psi\})$

In fact, the three implications above have proven stronger result than the equivalence. Note the universal quantification of pt and ps in (ii), and existential quantification in (iii). Composing (iii), (i), and (ii) in order, we obtain that $(\exists\ pt\ ps.\ t \overset{+}{\precsim}_{pt,ps} s\{\lambda\ \_\ \_.\ \Psi\})$ implies $(\forall\ (pt <_{\mathbb{I}} \top)\ (ps <_{\mathbb{I}} \top).\ t \overset{+}{\precsim}_{pt,ps} s\ \{\lambda\ \_\ \_.\ \Psi\})$. This means that FreeSim is irrelevant to the index when (i) there is no coinductive hypotheses (*i.e.*, coinductive proof has not started yet), and (ii) the postcondition is also irrelevant to the index. With IDX-MONO, we have the following:

THEOREM 5.5 (INDEX IRRELEVANCE). $\forall pt\ ps\ pt'\ ps'.\ t \overset{+}{\precsim}_{pt,ps} s\{\lambda\ \_\ \_.\ \Psi\} \Leftrightarrow t \overset{+}{\precsim}_{pt',ps'} s\{\lambda\ \_\ \_.\ \Psi\}$

This theorem is handy when proving metatheoretical properties like transitivity.

[5] ISim cannot replay ESim because there is no matching rule for STEP-SRC and STEP-TGT. These two rules unguard coinductive hypotheses, while the matching ones in ISim (STEP-SRC and STEP-TGT) do not.

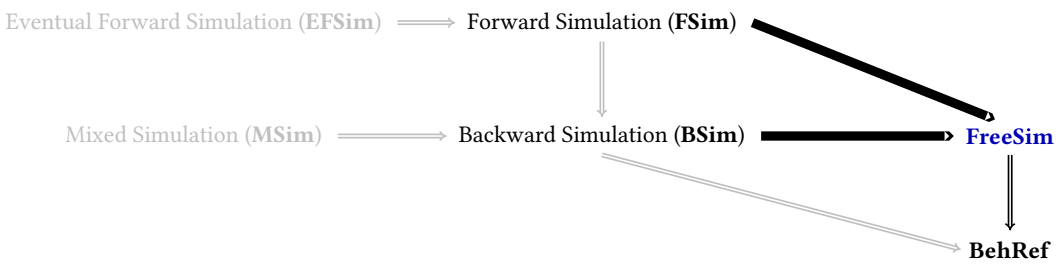


Fig. 12. Simulation techniques of CompCert (and CompCertM).

6 CASE STUDIES

We demonstrate the usefulness of FreeSim with two case studies. First, we apply FreeSim in CompCert (§6.1). Second, using FreeSim, we develop DTrees (Dual non-deterministic ITrees) library (§6.2).

6.1 Unifying Various Simulation Techniques in CompCert

Arguably, CompCert is one of the most realistic applications of formal verification techniques to date. CompCert is a multi-pass compiler, and to deal with different desiderata from different passes, CompCert and its variants employ a number of different simulations as shown in Fig. 12. In the second column are what we call "backbone" simulations of CompCert: FSim (Forward Simulation) and BSim (Backward Simulation). In the first column are more "advanced" simulations: EFSim (Eventual Simulation) and MSim (Mixed Simulation, developed in CompCertM [Song et al. 2019]). These advanced simulations are specialized simulations defined on top of backbone simulations by adding more features. All these simulations imply BehRef (behavioral refinement) shown at the bottom right, proven via transitively composing implications shown in the figure.

In this case study, we demonstrate that FreeSim could (i) simplify the adequacy proof of the above simulations, and also (ii) make it more *reusable*. We show this in two steps.

First, we show that both backbone simulations are replayable in FreeSim. As indicated by the definition of replayability, this proof is just a simple translation on a per-rule basis. In contrast, the implication proof from FSim to BSim necessitates a non-local translation of rules using a meticulously designed simulation relation.

Second, we show that both advanced simulations are replayable in FreeSim, but this time *reusing* backbone simulations and ISim via RPL-COIND-STEP. In fact, advanced simulations are no more than a direct mixture of backbone simulations and ISim, and they do not merit their own definition anymore.

In summary, we no more need the gray-colored definitions and implications in Fig. 12 since they are direct consequences of FreeSim and three arrows around it.

6.1.1 *Replaying Backbone Simulations.*

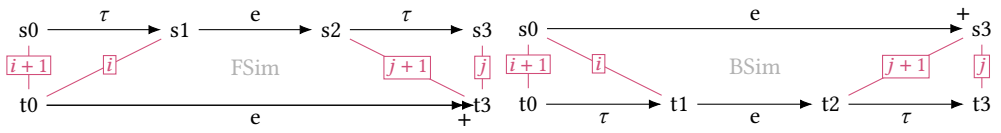
The core parts[6] of BSim and FSim are as follows:

$$\text{(BSim)} \quad \forall\, t\ \bar{\lesssim}_i\ s.\, \forall\, t \xhookrightarrow{e} t'.\, \exists\, s \xhookrightarrow{e}{}^{n} s'.\, \exists\, i'.\, (n = 0 \implies i' < i) \wedge t' \bar{\lesssim}_{i'} s'$$

$$\text{(FSim)} \quad \forall\, t\ \bar{\lesssim}_i\ s.\, \forall\, s \xhookrightarrow{e} s'.\, \exists\, t \xhookrightarrow{e}{}^{n} t'.\, \exists\, i'.\, (n = 0 \implies i' < i) \wedge t' \bar{\lesssim}_{i'} s'$$

where $\xhookrightarrow{e}{}^{n}$ means taking n consecutive steps while emitting an event e (possibly τ) and $\xhookrightarrow{e}{}^{n}_{\twoheadrightarrow}$ additionally requires that all the states it went through should be deterministic.[7] As the name suggests, BSim matches a given target step into multiple source steps and vice versa for FSim. Above two diagrams show typical proofs in FSim and BSim, respectively. FSim is handy when a single

[6] There are other conditions to ensure non-stuckness but we choose to ignore stuck states in this subsection for brevity.
[7] CompCert assumes the target state to be always deterministic, but we present this general form for consistency with MSim.



instruction in the source is compiled to multiple instructions in the target, but requires determinacy in the target. BSim is less convenient but could be used in non-deterministic languages.

As shown above, FSim and BSim have very different user interfaces, and even the implication proof from FSim and BSim involves highly non-trivial construction on the simulation relation.[8] Unlike propositional implication, replayability properly distinguishes the two as expected: *i.e.*, FSim implies BSim but BSim does not replay FSim. Even when the source and the target are both deterministic, FSim and BSim cannot simply replay each other. In FSim, in order to make a coinductive progress it must take a source step, and in BSim a target step. Thus, stuttering source steps in FSim (*e.g.*, from $t0 \mathrel{\bar{\precsim}_{i+1}} s0$ to $t0 \mathrel{\bar{\precsim}_{i}} s1$ on the left) cannot be replayed in BSim and, similarly, stuttering target steps in BSim cannot be replayed in FSim.

However, FreeSim can easily replay both FSim and BSim thanks to the asynchronous nature of FreeSim (§3.1). That is, both stuttering source steps and stuttering target steps are easily replayable in FreeSim. These replayability proof largely follow §3.3 and are omitted here for brevity.

6.1.2 Replaying Advanced Simulations. Now we see how we can replay EFSim and MSim using already replayable simulations. Core parts of these simulations are as follows:

$$\begin{aligned}
&\text{(MSim)} \quad \forall\, t \mathrel{\bar{\precsim}_{i}} s.\ (\forall\, t \xhookrightarrow{e} t'.\, \exists\, s \xhookrightarrow{e}{}^{n} s'.\ \exists\, i'.\, (n = 0 \implies i' < i) \wedge t' \mathrel{\bar{\precsim}_{i'}} s') \\
&\qquad\qquad \vee\ (\forall\, s \xhookrightarrow{e} s'.\, \exists\, t \xhookrightarrow{e}{}^{n} t'.\, \exists\, i'.\, (n = 0 \implies i' < i) \wedge t' \mathrel{\bar{\precsim}_{i'}} s') \\
&\text{(EFSim)} \quad \forall\, t \mathrel{\bar{\precsim}_{i}} s.\ \ \forall\, s \xhookrightarrow{e} s'.\, \exists\, t \xhookrightarrow{e}{}^{n} t'.\, \exists\, i'.\, (n = 0 \implies i' < i) \wedge \exists\, m.\, \forall\, s' \xhookrightarrow{\tau}{}^{m} s''.\, t' \mathrel{\bar{\precsim}_{i'}} s''
\end{aligned}$$

Note that MSim is merely a mixture of FSim and BSim: for each given state, one can choose whether to use FSim or BSim. However, note also that its implication proof (to BSim) is non-trivial. Especially, despite the innate similarity between the implication proof from FSim to BSim and MSim to BSim, it is unclear how to reuse the proof of the former in the latter. Indeed, CompCertM proves the latter from scratch, despite many overlaps with the former.

With FreeSim, this situation can be improved a lot. Since FreeSim can replay both FSim and BSim with the same embedding, MSim is also replayable (with the same embedding) by RPL-COIND-STEP. This makes both the definition and implication proof of MSim obsolete.

Similarly, EFSim could be seen as a mixture of FSim and ISim. It proceeds the same with FSim, but at the end, it takes few more stuttering steps in the source while *not* decrementing the stuttering index, as in ISim. Thus, EFSim is also easily replayable in FreeSim.

We conclude this subsection with a remark that these two advanced simulations are only two instances of possible mixtures of backbone simulations: we can come up with many more, such as mixing ISim with BSim (EBSim?), mixing EFSim and EBSim (EMSim?), or adding *target* stuttering steps at the end of execution in EFSim (EEFSim?). We believe FreeSim and its replayability results are a meaningful first step to show how to modularly derive different variants of simulations instead of defining and proving them in an ad-hoc way. It will be an interesting future work to further refine and develop theories around replayability.

6.2 DTrees: ITrees With Dual Non-Determinism

In the context of program refinement, it is common to write abstract, mathematical specifications of the given system in STS directly in terms of states and transitions [Gu et al. 2015; Song et al. 2019].

[8]The proof is available in CompCert codebase [Leroy 2023].

$\mathrm{k} \in \mathrm{ktree}\ A\ B \qquad \mathrm{ktree}\ A\ B \triangleq A \to \mathrm{dtree}\ B$

$\mathrm{d} \in \mathrm{dtree}\ R \overset{\mathrm{coind}}{=} \mathtt{Ret}(r \in R) \mid \mathtt{Tau}(\mathrm{d}) \mid \mathtt{Eff}(X \in \mathbf{Set},\ e \in E\ X,\ \mathrm{k} \in \mathrm{ktree}\ X\ R)$ constructors in ITrees
$\quad \mid \exists(X \in \mathbf{Set},\ \mathrm{k} \in \mathrm{ktree}\ X\ R) \mid \forall(X \in \mathbf{Set},\ \mathrm{k} \in \mathrm{ktree}\ X\ R)$ new ones in DTrees

$(hd \in \mathrm{ktree}\ A\ B \ggg tl \in \mathrm{ktree}\ B\ C) \in \mathrm{ktree}\ A\ C \triangleq \lambda a.\ (hd\ a \gg\!= tl)$

$$(hd \in \mathrm{dtree}\ A \gg\!= tl \in \mathrm{ktree}\ A\ B) \in \mathrm{dtree}\ B \overset{\mathrm{coind}}{=} \begin{cases} \text{If } hd = \mathtt{Ret}(a) & : \ tl\ a \\ \text{If } hd = \mathtt{Tau}(\mathrm{d}) & : \ \mathtt{Tau}(\mathrm{d} \gg\!= tl) \\ \text{If } hd = \mathtt{Eff}(X, e, \mathrm{k}) & : \ \mathtt{Eff}(X, e, \mathrm{k} \ggg tl) \\ \text{If } hd = \exists(X, \mathrm{k}) & : \ \exists(X, \mathrm{k} \ggg tl) \\ \text{If } hd = \forall(X, \mathrm{k}) & : \ \forall(X, \mathrm{k} \ggg tl) \end{cases}$$

$\mathrm{k}^{\mathrm{op}} \triangleq \lambda a.\ (\mathrm{k}\ a)^{\mathrm{op}}$

$$\mathrm{d}^{\mathrm{op}} \overset{\mathrm{coind}}{=} \begin{cases} \text{If } \mathrm{d} = \mathtt{Ret}(a) & : \ \mathtt{Ret}(a) \\ \text{If } \mathrm{d} = \mathtt{Tau}(\mathrm{d}') & : \ \mathtt{Tau}(\mathrm{d}'^{\mathrm{op}}) \\ \text{If } \mathrm{d} = \mathtt{Eff}(X, e, \mathrm{k}) & : \ \mathtt{Tau};\ \mathtt{Eff}(X, e, \mathrm{k}^{\mathrm{op}}) \\ \text{If } \mathrm{d} = \forall(X, \mathrm{k}) & : \ \mathtt{Tau};\ \exists(X, \mathrm{k}^{\mathrm{op}}) \\ \text{If } \mathrm{d} = \exists(X, \mathrm{k}) & : \ \mathtt{Tau};\ \forall(X, \mathrm{k}^{\mathrm{op}}) \end{cases}$$

$\mathtt{iter}\ (\mathrm{k} \in \mathrm{ktree}\ I\ (I + R)) \in \mathrm{ktree}\ I\ R \triangleq \ldots$

$E \rightsquigarrow F \triangleq \forall X.\ E\ X \to F\ X$
$\mathtt{interp}\ (\mathcal{H} \in E \rightsquigarrow \mathrm{dtree}_F) \in \mathrm{dtree}_E \rightsquigarrow \mathrm{dtree}_F \triangleq \ldots$

$\mathtt{stateT}_S\ M\ X \triangleq S \to M\ (S \times X)$
$\mathtt{interp\_stateT}\ (\mathcal{H} \in E \rightsquigarrow \mathtt{stateT}_S\ \mathrm{dtree}_F) \in$
$\quad \mathrm{dtree}_E \rightsquigarrow \mathtt{stateT}_S\ \mathrm{dtree}_F \triangleq \ldots$

Fig. 13. Core definitions of DTrees

Note that we do not want to write these abstractions with traditional programming languages which contaminates abstractions with too much details (*e.g.*, checking integer overflow). However, writing STS directly turns out to be too tedious and makes it rather hard to reason about.

This situation has been greatly improved with the advent of ITrees, which now serves as a perfect specification language in multiple projects [Sammler et al. 2023; Song et al. 2023]. In this context, ITrees could be seen as a more enriched formulation of STS accompanied with (bi)simulations, rich combinators and (equational) theories around them. These combinators and theories greatly simplify writing and reasoning about mathematical STS. Specifically, in ITrees many of the verification could be carried out using elementary *rewrite*s (utilizing **transitivity** of the simulation).

However, the potential of ITrees have not been fully utilized yet in this context. The problem is that ITrees does not come up with native support for non-determinism. While the definitions of ITrees could easily be extended to model (dual) non-determinism (done in previous works [Lee et al. 2023; Song et al. 2023]), having desired simulation and theories (*esp.*, transitivity) for such a definition is challenging and they were missing. Specifically, we want (directed) simulation (*not* bisimulation, which is too restrictive in this context) for dual non-determinstc ITrees and theories for it in the style of ITrees.

That is precisely the goal of this section: we develop DTrees library providing such. This result is made possible largely due to the simpler formulation of simulation with FreeSim (§3.4 and §5). Proving properties like transitivity is quite involved and the simpler formulation made it down to a manageable level of complexity. Moreover, since our simulation is based on FreeSim, it naturally enjoys all the benefits of FreeSim (§3) which should be useful to the user of this library.

Definitions. As mentioned, one way to understand ITrees is to think of it as another formulation of (determinstc) STSs that has additional structure (user-defined *effects*), rich combinators, and equational theories around them. Specifically, an itree of type "$\mathrm{itree}_E\ R$" is parameterized over effects type E and return type R; in general E can contain arbitrary user-defined algebraic effects (*e.g.*, `put/get` operators for accessing states), and can also contain visible events $\mathcal{E}$ of STS. In this understanding, all user-defined effects will eventually be handled (thus removed) with user-defined handlers and will leave "$\mathrm{itree}_{\mathcal{E}}\ R$", which correspond [Song et al. 2023] to deterministic STS.

DTrees add dual non-determinism to ITrees, and thus can represent STS with dual non-determinism. DTrees are defined on top of ITrees to directly piggy-back on existing combinators (importantly, `iter` and `interp`) and theories. That is, "dtree$_E$ R" is defined simply as "itree$_{E+nondetE}$ R" where *nondetE* comprises effects representing dual non-determinism, not visible events.

We provide the definition of dtree, unfolded, in Fig. 13, and we implicitly use an effect type E everywhere (unless explicitly written) hereafter. Then, a dtree is coinductively defined with five constructors each corresponding to these "sort"s in STS. First three constructors are inherited from ITrees: (i) `Ret` case for returning with a return value, (ii) `Tau` case for a silent step to the next state (dtree), (iii) `Eff` case for an effectful step to the next state, and second two constructor are newly added in DTrees: (iv) ∃ case for demonic non-determinism, (v) ∀ case for angelic non-determinism. Semantically, `Tau` constructor is equivalent to (iv) or (v) given X as a unit, but is still needed to piggy-back on ITrees. Following DimSum [Sammler et al. 2023], we use ∃ notation for demonic case since if there *exists* $x \in X$ such that the continuation (k x) has certain trace, then it is considered a trace of $\exists(X,\ \mathrm{k})$, and vice versa for ∀.

Then, the monadic bind operator (≫=) for DTrees is defined in Fig. 13. For a given dtree *hd* and its continuation *tl*, bind operator computes *hd* and if it terminates with `Ret` case, continues execution with *tl* with the return value. We use standard punctuation notation for ∃/∀ cases, and standard monadic notations (including ; for sequential composition) for DTrees. For instance, the following dtree represents a computation that either returns minus one or zero:

$$\exists b \in \mathbb{B}.\ \texttt{Tau};\ r \leftarrow (\textbf{if}\ b\ \textbf{then}\ \texttt{Ret}(1)\ \textbf{else}\ \texttt{Ret}(0));\ \texttt{Ret}(-r)$$

The dualize operator, $-^{\mathrm{op}}$, simply swaps demonic and angelic non-determinism of the given dtree (added `Tau`s are consequences of piggy-backing on ITrees). `iter` and `interp` are the core combinators from ITrees. `iter` computes (possibly infinitely) the given computation, k, until it returns (*i.e.*, returning a value with **`right`**). `interp` translates a dtree with effect type E into that with effect type F by applying the provided handler $\mathcal{H}$ for every effect invocations. `interp_state` is a more advanced version where $\mathcal{H}$ could be stateful, as manifested with the use of standard state transformer, `stateT`. For an example, the following dtree represents a computation that repeatedly prints "42" for indefinite (possibly infinite) amount of times.

$$(\texttt{iter}\ (\lambda\_.\ \texttt{Eff}(\texttt{Print}(42));\ \forall b \in \mathbb{B}.\ \textbf{if}\ b\ \textbf{then}\ \texttt{Ret}(\textbf{left}\ ())\ \textbf{else}\ \texttt{Ret}(\textbf{right}\ ())))^{\mathrm{op}}$$

Finally, there is a denotation operator $[\![-]\!]$ that maps a dtree$_{\mathcal{E}}$ into a STS [Song et al. 2023]. Moreover, this operator is *surjective*: meaning that dtree$_{\mathcal{E}}$ is at least as expressible as STS.

Simulation and adequacy. We use $\overset{+}{\gtrapprox}$ notation for simulation for dtrees. Its definition directly follows §4 (except that event case is extended to arbitrary effect E, not just $\mathcal{E}$) and is omitted for brevity. Indeed, the adequacy result directly relates $\overset{+}{\gtrapprox}$ with $\overset{+}{\gtrsim}$.

Theorem 6.1 (Lifting). *For DTrees d_t, d_s of type dtree$_{\mathcal{E}}$ and a postcondition Ψ, the following holds:*

$$d_t \overset{+}{\gtrapprox} d_s\{\lambda\ \_\ \_.\ \Psi\} \implies \mathrm{init}\ [\![d_t]\!] \overset{+}{\gtrsim} \mathrm{init}\ [\![d_s]\!]\{\lambda\ \_\ \_.\ \Psi\}$$

Corollary 6.2 (Adequacy). *For a pair of DTrees d_t, d_s of type dtree$_{\mathcal{E}}$, we have:*

$$d_t \overset{+}{\gtrsim} d_s \implies [\![d_t]\!] \sqsubseteq_{beh} [\![d_s]\!]$$

Theories. Now we present the selected core theories for DTrees given in Fig. 14. Our rules largely follow the style of ITrees (where its bisimulation, ≈, is changed to $\overset{+}{\gtrapprox}$) unless explicitly mentioned.

EUTT and EUTTGE are rules that allow us to piggy-back on a large body of theories in original ITrees. EUTT allows rewriting with ≈ where $a \approx b$ means that a and b have equal structure (everything except `Tau`) but can differ in finite number of `Tau`s in between. To put it simply, EUTT allows attaching/removing finite number of `Tau`'s for a proven simulation. This is useful since combinators

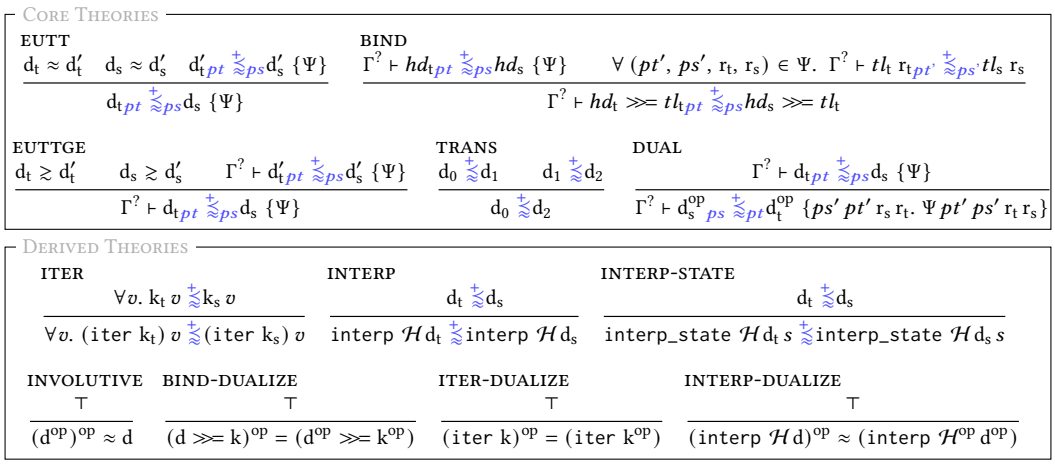


Fig. 14. Selected theories of DTrees

in ITrees automatically adds Taus (as seen in the definition of $-^{\mathrm{op}}$) and we want to ignore them in our reasoning. Note that this rule cannot be applied when there is any coinductive hypothesis: doing so is unsound (it gives free progress by attaching Taus). EUTTGE is a similar rule but meant to be used in the middle of the coinductive proof. However, attaching Taus is rightfully forbidden: it only allows removing Taus ($a \gtrsim b$ means b has less Taus).

These two rules allow migrating all the theories in the form of a simple equation (*e.g.*, $a \approx b$), but some theories where simulation appears in both positive and negative position need to be newly proven for $\overset{+}{\gtrapprox}$. At the core of these rules are only three: BIND, TRANS, and DUAL (replacing symmetry). BIND is stronger than the one in ITrees in that it allows passing partial progress between split computations (§3.2). TRANS is the most challenging one to prove: the proof requires case-analysis on both premises which basically results in cases with quadratic the number of constructors. FreeSim has a much simpler definition compared to equivalent ones (§3.4 and §5) with fewer number of constructors and this was crucial for our proof. The symmetry rule, which simply flips the source and the target, does not hold anymore since we are using directed simulation. Such a rule is adjusted to DUAL rule that additionally takes dual on both sides. It is easy to see that DUAL rule holds.

Finally, we have rules that are derived from core theories of ITrees and DTrees. Rules ITER, INTERP, and INTERP-STATE are direct translations of corresponding rules in ITrees where $\approx$ is changed into $\overset{+}{\gtrapprox}$. These rules are easily derived from BIND above. Finally, equational theories for $-^{\mathrm{op}}$, which is defined using `interp`, are directly derived from equational theories for `interp` in ITrees.

7 DISCUSSION AND RELATED WORK

In §7.1, we discuss how one may push the boundaries of existing approaches (explicit/implicit stuttering) and how that compares with our interface, FreeSim. In §7.2, we discuss related work.

7.1 Pushing Boundaries of Existing Approaches

There are clever tricks to extend the boundary of ISim/ESim, and we compare them with FreeSim.

Adding "skip"s everywhere in implicit stuttering. The most important difference between implicit stuttering and FreeSim is that the latter produces more (partial) coinductive progress. Thus, it would be possible to mitigate the limitations of implicit stuttering by adding **`skip`**s in between every instruction. For example, in REORDER, if we had **`skip`** in between `x` and `y`, one could prove the example with implicit stuttering. Since inserting **`skip`**s manually is prohibitively cumbersome,

we could instead add them automatically to an arbitrary STS as follows:

$$\begin{array}{l}\text{AddSkips}(\mathbb{S} \in \mathit{STS}) \in \mathit{STS} \triangleq \{X_{\mathbb{S}} \times \mathbb{B},\ \{(x,\top) \xhookrightarrow{\tau} (x,\bot) \mid x \in X_{\mathbb{S}}\} \uplus \{(x,\bot) \xhookrightarrow{e} (y,\top) \mid x \xhookrightarrow{e}_{\mathbb{S}} y\},\\ \qquad\qquad (\text{init}_{\mathbb{S}},\bot),\ \lambda(x \in X, b \in \mathbb{B}).\ \textbf{if}\ b == \bot\ \textbf{then}\ \text{sort}_{\mathbb{S}}\ \textbf{else}\ \text{Tau}\ \}\end{array}$$

The AddSkips operator takes an STS, $\mathbb{S}$, and returns an STS with **skip**s inserted. That is, each state $x_{\mathbb{S}}$ gets divided into $(x_{\mathbb{S}},\top)$, $(x_{\mathbb{S}},\bot)$, where the former corresponds to the **skip**—it has a single outgoing edge with τ into the later—and the latter corresponds to the original state $x_{\mathbb{S}}$.

With this, one can prove a meta-theoretic result that AddSkips preserves the behavior (*i.e.*, $\text{AddSkips}(\mathbb{S}) \equiv_{\text{beh}} \mathbb{S}$). Then, given a goal $\mathbb{T} \sqsubseteq_{\text{beh}} \mathbb{S}$, we can turn it into $\text{AddSkips}(\mathbb{T}) \sqsubseteq_{\text{beh}} \text{AddSkips}(\mathbb{S})$ and now it suffices to prove an easier goal, $\text{AddSkips}(\mathbb{T}) \preccurlyeq \text{AddSkips}(\mathbb{S})$.

We can push this even further by inserting more than one **skip**: given a well-founded order $\mathbb{O}$ with $\bot$ and $\top$, we now split each state $x_{\mathbb{S}}$ into $(x_{\mathbb{S}}, o \in \mathbb{O})$:

$$\begin{array}{l}\text{AddManySkips}(\mathbb{S}) \triangleq \{X_{\mathbb{S}} \times \mathbb{O},\ \{(x,o_1) \xhookrightarrow{\tau} (x,o_0) \mid x \in X_{\mathbb{S}} \wedge o_0 <_{\mathbb{O}} o_1\} \uplus \{(x,\bot) \xhookrightarrow{e} (y,\top) \mid x \xhookrightarrow{e}_{\mathbb{S}} y\},\\ \qquad\qquad (\text{init}_{\mathbb{S}},\bot),\ \lambda(x \in X, b \in \mathbb{B}).\ \textbf{if}\ b == \bot\ \textbf{then}\ \text{sort}_{\mathbb{S}}\ \textbf{else}\ \text{Tau}\ \}\end{array}$$

And this results in the same interface as FreeSim. Specifically, $t \overset{+}{\underset{pt}{\preccurlyeq}}_{ps} s$ corresponds to $(t, pt) \preccurlyeq (s, ps)$: *i.e.*, progress indices correspond to the number of free **skip**s remaining in the current state. Check that (i) executing an actual stuttering step to the next state $(x,\bot) \xhookrightarrow{e} (y,\top)$ refreshes the skip-count to $\top$, behaving the same as the stuttering rules in FreeSim, and (ii) executing free **skip**s on both the target and the source $(x,o_1) \xhookrightarrow{\tau} (x,o_0)$ decrements skip-counts and makes coinductive progress, behaving the same as COIND-PROG. Thus, ISim after pre-processing with AddManySkips could be seen as another implementation of the FreeSim interface.

This observation could potentially be useful in a project where one already has adequacy proof for an implicit stuttering and wants to reuse it. On the other hand, adding such **skip**s could potentially cause cluttering through the whole development and would be infeasible in some projects. For instance, the whole theory of ITrees is proven to be agnostic to the number of **skip**s (*i.e.*, respecting $\approx$), but doing so required significant technical effort.

Hiding stuttering index in explicit stuttering. The most important limitation of explicit stuttering is that the user needs to set up the right index upfront. It would be possible to mitigate this limitation by hiding the index behind existential quantifiers: $t \overset{-}{\preccurlyeq}' s \triangleq \exists i.\ t \overset{-}{\preccurlyeq}_i s$. This definition will indeed enjoy most of the important meta-theoretic properties (*e.g.*, BIND in §6).

However, this trick has one serious problem: existing coinductive techniques do not support an existential quantifier in the goal. Thus, for a proof that requires manual coinductive reasoning, this abstraction does not work and one needs to fall back to the original definition of explicit stuttering.

7.2 Related Work

Termination-insensitive refinement. When one aims for a lower goal, often called *termination-insensitive refinement* [Frumin et al. 2018; Turon et al. 2013], more elementary simulations suffice. Termination-insensitive refinement basically interprets silent divergence in the target as an empty behavior, meaning that it allows a non-terminating program in the target to refine an arbitrary program in the source. For such a weaker result, BSim without stuttering index (equivalently, ISim where asynchronous target step makes coinductive progress) suffices. However, such an elementary definition would only work for termination-insensitive refinement, and sophisticated stuttering is a necessary evil when one wants a stronger result like termination-sensitive refinement.

Simuliris. Simuliris [Gäher et al. 2022] uses an implicit stuttering simulation, as well as a clever trick that works for their programming language to side step all the complexities coming from coinductive reasoning. In the language they consider, the only sources of infinite execution are

while loops and recursion, and they have focused on specialized support for these. Specifically, (i) rules for executing **while** loops are restricted to synchronous execution in both sides only, and (ii) they utilize the fact that every iteration of **while** loop starts with an additional "unfolding" step. These two together give a simple rule for **while** that does not require users to show coinductive progress explicitly, since progress is already made at the beginning of the loop.

While this trick greatly simplifies the verification and works in many scenarios, it does not work for: (i) examples that need coinductive hypotheses out of these synchronous **while** loops (*e.g.*, REORDER), and (ii) languages that do not have such an *unfolding* step in the loop semantics (*e.g.*, CCS or the RTL language in CompCert).

Simulations for dual non-determinism. CCR [Song et al. 2023] originally defined a simulation (for termination-sensitive refinement) with explicit stuttering index and inherits its limitations. CCR also defines a simulation on dtree, but does not provide the metatheory (*esp.*, transitivity) that we offer. DimSum [Sammler et al. 2023] also defines a simulation for STS with dual non-determinism, but it aims for termination-insensitive refinement and has the aforementioned elementary definition of simulation. DimSum's simulation does not satisfy DUAL, and this seems to be a fundamental limitation for such elementary simulations.

Choice Trees. Choice Trees [Chappe et al. 2023] (abbreviated as CTrees) are a variant of ITrees supporting non-determinism. CTrees provide bisimulation, and DTrees provide directed simulation, which is fundamentally different with respect to stuttering. Bisimulation, even the *termination-sensitive* one, could be defined as a conjunction of two directed elementary simulations, whereas (as mentioned) sophisticated stuttering is essential in the latter. Also, CTrees at the moment have limited support for weak bisimulation (lacking proper up-to techniques for the bind operator) unlike us (BIND). Other (minor) differences are that we support both angelic and demonic non-determinism while CTrees only consider demonic. On the other hand, CTrees distinguishes between steps that can cause divergence and steps that cannot. The datatype we use in DTrees is defined on top of ITrees, whereas CTrees define a separate datatype inspired by ITrees.

Bisimulation. The idea of tracking the guardedness asynchronously could be applicable in domains outside directed simulation. Specifically, the weak bisimulation in ITrees (`eutt`) is defined in the style of implicit stuttering, and we believe progress indices could be employed in that setting.

Diacritical progress approaches [Biernacki et al. 2019a,b,c] have decomposed a bisimulation as a *conjunction* of a part about computational steps and a part about administrative steps, and accordingly distinguished between strong and weak up-to techniques. Since a stuttering simulation is a *disjunction* of stuttering cases and progressive cases, a diacritical progress approach is not directly applicable. Nevertheless, it would be an interesting future work to explore ways of combining diacritical progress with our approach.

ACKNOWLEDGMENTS

Minki Cho and Dongjae were supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT2102-03. Lennard Gäher was supported by an Amazon Research Award.

DATA AVAILABILITY STATEMENT

The Coq formalization of this paper may be found at [Cho et al. 2023].

REFERENCES

Ralph-Johan Back and Joakim Wright. 2012. *Refinement calculus: a systematic introduction.* Springer Science & Business Media. https://dl.acm.org/doi/10.5555/551462

Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. 2019a. Bisimulations for delimited-control operators. *Logical methods in computer science* 15 (2019).

Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. 2019b. Diacritical companions. *Electronic Notes in Theoretical Computer Science* 347 (2019), 25–43.

Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. 2019c. Proving soundness of extensional normal-form bisimilarities. *Logical Methods in Computer Science* 15 (2019).

Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proc. ACM Program. Lang.* 7, POPL, Article 61 (jan 2023), 31 pages. https://doi.org/10.1145/3571254

Minki Cho, Youngju Song, Dongjae Lee, Lennard Gäher, and Derek Dreyer. 2023. *Stuttering for Free (OOPSLA 2023 Artifact).* https://doi.org/10.5281/zenodo.8331740

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A mechanised relational logic for fine-grained concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science.* 442–451.

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015).*

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13).* Association for Computing Machinery, New York, NY, USA, 193–206. https://doi.org/10.1145/2429069.2429093

Jérémie Koenig and Zhong Shao. 2020. Refinement-Based Game Semantics for Certified Abstraction Layers. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20).* Association for Computing Machinery, New York, NY, USA, 633–647. https://doi.org/10.1145/3373718.3394799

Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur. 2023. Fair Operational Semantics. *Proc. ACM Program. Lang.* 7, PLDI, Article 139 (jun 2023), 24 pages. https://doi.org/10.1145/3591253

Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006).*

Xavier Leroy. 2023. Implication proof from forward simulation to backward simulation. https://github.com/AbsInt/CompCert/blob/35feefd229792e6b560ccf156465a6e309bc1d98/common/Smallstep.v#L1612

Jacob R Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R Wilcox, and Xueyuan Zhao. 2020. Armada: low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 197–210. https://doi.org/10.1145/3385412.3385971

Robin Milner. 1971. An Algebraic Definition of Simulation Between Programs. In *International Joint Conference on Artificial Intelligence.* https://dl.acm.org/doi/abs/10.5555/1622876.1622926

Robin Milner. 1989. *Communication and concurrency.* Vol. 84. Prentice hall Englewood Cliffs.

Kedar S. Namjoshi. 1997. A Simple Characterization of Stuttering Bisimulation. In *Foundations of Software Technology and Theoretical Computer Science, 17th Conference, Kharagpur, India, December 18-20, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1346)*, S. Ramesh and G. Sivakumar (Eds.). Springer, 284–296. https://doi.org/10.1007/BFb0058037

Damien Pous. 2016. Coinduction all the way up. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science.* 307–316. https://doi.org/10.1145/2933575.2934564

Damien Pous and Davide Sangiorgi. 2011. *Enhancements of the bisimulation proof method.* Cambridge University Press, 233289. https://doi.org/10.1017/CBO9780511792588.007

Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2019. The high-level benefits of low-level sandboxing. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32. https://doi.org/10.1145/3371100

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-Language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (jan 2023), 31 pages. https://doi.org/10.1145/3571220

Davide Sangiorgi. 1998. On the bisimulation proof method. *Mathematical Structures in Computer Science* 8, 5 (1998), 447–479.

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371091

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (jan 2023), 31 pages. https://doi.org/10.1145/3571232

The Coq Development Team. 2021. The Coq Proof Assistant 8.13.2 Reference Manual. https://coq.github.io/doc/V8.13.2/refman/.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. 377–390. https://doi.org/10.1145/2500365.2500600

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371119

Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An Equational Theory for Weak Bisimulation via Generalized Parameterized Coinduction. *CoRR* abs/2001.02659 (2020). arXiv:2001.02659 http://arxiv.org/abs/2001.02659

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 427–440. https://doi.org/10.1145/2103656.2103709

Received 2023-04-14; accepted 2023-08-27