A Family of Sims with Diverging Interests

NICOLAS CHAPPE, CNRS, Univ. Grenoble Alpes, Grenoble INP, VERIMAG, France

Simulations are widely-used notions of program refinement. This paper discusses and compares several of them, in particular notions of simulation that are both weak and sensitive to divergence. Complex simulation proofs performed in proof assistants, for instance in a verified compilation setting, often rely on variants of normed simulation, which is not complete with respect to divergence-sensitive weak simulation. We propose to bridge this gap with μdiv-simulation, a novel notion of simulation that is equivalent to classical divergence-sensitive weak simulation, and designed to be as comfortable to use as modern characterizations of normed simulation. We then define a parameterized notion of simulation that covers strong simulation, weak simulation, μdiv-simulation, and 9 more notions of simulation, and jointly establish various "up-to" reasoning techniques for these 12 notions. Our results are formalized in Rocq and instantiated on two case studies: Choice Trees and a CompCert pass. Verified compilation is a major motivation for our study, but because we work with an abstract LTS setting, our results are also relevant to other fields that make use of divergence-sensitive weak simulation, such as model checking.

CCS Concepts: • **Theory of computation → Program reasoning**; *Concurrency*; • **Software and its engineering → Semantics**.

Additional Key Words and Phrases: Rocq, simulation, verified compilation, coinduction up-to

1 Introduction

Notions of program comparison are pervasive in programming language theory: they are useful to compare a model with its implementation in model checking, to reason about process calculi in concurrency theory, or to prove the correctness of program transformations in verified compilation. *Simulations* are an interesting class of notions of program comparison, with the valuable property that they typically enable *local* reasoning about programs.

Roughly speaking, a program simulates another one if the former can reproduce every step of computation of the latter. There actually exist many kinds of simulation relations and bisimulation relations [44],[1] with numerous applications in concurrency theory [36] and model checking [4, 17].

The present paper studies and compares several notions of simulation, with a particular focus on the usabilty of these notions in the setting of a proof assistant. The field of *verified compilation* largely relies on proofs of program comparison in proof assistants, and will serve as a major motivation throughout this paper. The aim of verified compilation is to formally verify the correctness of compilers and program transformations in order to prevent miscompilation bugs. This field of research has been rapidly developing these past few decades, a notable achievement being CompCert [22], a formally verified optimizing compiler from C to several architectures, with industrial users.

As any compiler, a verified compiler performs a series of transformations on a source program in order to optimize it and to compile it down to assembly. But unlike a regular compiler, a verified compiler comes with a *proof* that each of these passes preserves the semantics of the program. More specifically, the developer of a verified compilation pass seeks a result of *behavioral inclusion* (or *refinement*): the behaviors of the program resulting from a compilation pass should all be

[1]Bisimulations are notions of behavioral equivalence closely related to simulations.

Author's Contact Information: Nicolas Chappe, CNRS, Univ. Grenoble Alpes, Grenoble INP, VERIMAG, Grenoble, France, nicolas.chappe@cnrs.fr.

possible behaviors of the program given as input to the pass. A full verified compiler thus requires *many* proofs of behavioral inclusion (one for each pass), that can ultimately be composed into an end-to-end proof of correctness. This notion of behavioral inclusion is formally captured by *trace inclusion*, but trace inclusion can be a bit unwieldy, so verified compilation projects rather stick to *simulation* [28], a more restrictive notion of behavioral inclusion that implies trace inclusion. One thus seeks to establish that the source program simulates the associated compiled program.[2]

The pervasiveness of non-trivial interactive simulation proofs in verified compilers justifies the need for clean and powerful reasoning principles around simulations mechanized in a proof assistant. It turns out that one notion fits most of the needs of the field: *divergence-sensitive weak simulation*. It is called *weak* because this kind of simulation has a notion of internal "tau" events that carry unobservable semantic information that can be hidden, and *divergence-sensitive* because silent divergence (e.g., empty infinite `while` loops) should not be introduced by a correct compiler. Section 2.2 introduces it more formally, as it constitutes the main object of study of the present paper.

Extensive studies about notions of simulation (and bisimulation) are often motivated by process algebras. Weak (bi)simulation is particularly relevant in this setting and has thus been thoroughly studied. Divergence sensitivity has also been studied in this community [42, 43, 46],[3] but with a particular focus on pen-and-paper results of bisimulation, for which the stakes can be different from mechanized simulation proofs. Challenges with the handling of divergence sensitivity in mechanized proofs were precisely the motivation for the introduction of *normed simulation* [18, 30], a notion that comes from the model checking community and has then been widely adopted in the verified compilation community.

Normed simulation is a notion of weak simulation that ensures divergence preservation through the use of a well-founded measure on program states, with the drawback that it is not complete with respect to divergence-sensitive weak simulation. Section 2.3 introduces it more precisely. It is pervasive in CompCert, and in other verified compilation projects [15, 48], Despite the ubiquity of normed simulation, tools for proving simulation results have significantly evolved over the years. In particular, three research directions from the 2010s arguably constitute major milestones for simulation proof techniques:

- *Coinduction up-to*, a powerful approach to simulation proofs, has become more usable in proof assistants thanks to the development of novel theories and accompanying libraries [19, 33, 38]. We illustrate this powerful approach in Section 2.4.
- A modern characterization of normed simulation has emerged [29] and is now widely used [15, 16, 47]. It no longer requires explicitly providing a well-founded measure, but instead relies on a novel mixed *inductive-coinductive* characterization (further discussed at the end of Section 2.3).
- The CompCertTSO project [45], which successfully developed a verified compiler for *concurrent* C programs, could not prove that normed simulation holds for one of their passes (a fence elimination pass), and consequently developed a novel notion dubbed *weak-tau simulation* [41] (further discussed in Section 7). This notion is based on mutually-defined relations of simulation and divergence preservation. It is sound with respect to divergence-sensitive weak simulation and sufficiently complete to prove their refinement result of interest.

[2]A result in this direction is often called a "backward" simulation. Unless otherwise noted, all the notions discussed in this paper are primarily meant to be used in this direction.
[3]Note however that divergence sensitivity is not always desired: process algebras sometimes assume the property of *fair abstraction from divergence* that makes divergence sensitivity irrelevant [2].

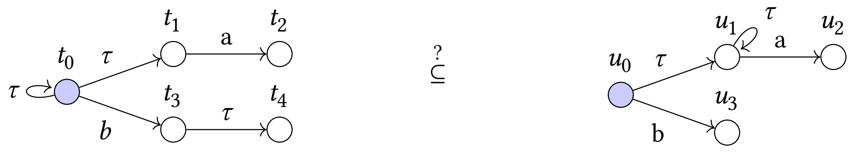


Fig. 1. A behavioral inclusion result that cannot be proved with divergence-sensitive simulation techniques currently used

These three independent advances are helpful to ease proofs of simulation in a mechanized setting. Unfortunately, to our knowledge, these ideas have still not been combined into a single notion of simulation, which is of much consequence. Consider for instance the LTS states t_0 and u_0 depicted in Figure 1. State t_0 is simulated by u_0 according to classical divergence-sensitive weak simulation (as defined in Section 2.2), but neither variants of normed simulation nor weak-tau simulation are sufficiently complete to make it provable, as we explain in their respective sections. This result is simply unprovable with the tools of existing verified compilation projects.

The main ambition of the present paper is to bridge this gap by combining the three recent advances discussed above into a novel notion of simulation dubbed *μdiv-simulation* and to formalize it in the Rocq prover [40]. Indeed, μdiv-simulation ensures divergence preservation through two mutually dependent coinductive relations, one of which is inductive-coinductive, and we develop powerful reasoning principles around it thanks to coinduction up-to, thus combining the three aforementioned elements. Unlike most existing notions of simulation for verified compilation, μdiv-simulation is sound and complete with respect to classical divergence-sensitive weak simulation. Furthermore, it arguably enjoys more powerful reasoning principles than existing notions of normed simulation thanks to the use of coinduction up-to. The simulation from Figure 1 is proved using μdiv-simulation in Section 3.5.

Restricting our study to a flavor of divergence-sensitive weak simulation would not be entirely satisfying, because there is a whole family of *sims* relevant to programming language theory. First, beyond divergence sensitivity, deadlock sensitivity is also worthy of interest in verified compilation. Deadlock-sensitive notions of simulation prevent the introduction of deadlocks in the compiled program by the compiler, which is especially relevant in a concurrent setting, in particular for CompCertTSO [45]. But deadlock sensitivity is not always desired, as some memory models exploit deadlock insensitivity to handle out-of-memory errors [5]. Second, other notions of simulation can be useful as proof intermediates for divergence-sensitive weak simulation, in particular strong simulation. At the very least, our study should cover divergence-sensitive weak simulation, deadlock-sensitive divergence-sensitive weak simulation, strong simulation, and deadlock-sensitive strong simulation. But this is not the whole story yet: the literature about *bi*simulations defines some notions between strong bisimulation and weak bisimulation that can be valuable as proof intermediates, such as *expansions* [37]. One can expect similar results to hold for variants of weak simulation and thus for divergence-sensitive weak simulation, further raising the number of notions of simulation of interest. As a consequence, Section 4 generalizes μdiv-simulation to a parameterized notion of simulation, with two boolean parameters and one ternary parameter, and we jointly study the 12 resulting notions of simulation (including μdiv-simulation). This idea of jointly studying several notions of program comparison through a parameterized definition has already been explored for weak bisimulation [49], but only in a deterministic setting.

Throughout this paper, clickable references to our Rocq development [11] are marked with the Rocq logo [`rocq-sims`]. While verified compilation is a major motivation for the present work, our results and their Rocq formalization only depend on an abstract setting of LTS, and can thus be instantiated to problems in other fields that have uses for divergence-sensitive weak simulation, such as model checking [4].

Structure of this paper. Section 2 gives important context for this work. It states the definition of relevant existing notions of simulation and demonstrates the use of coinduction up-to in simulation proofs with an example. Section 3 defines µdiv-simulation, validates it against divergence-sensitive weak simulation, and proposes an example of a simulation proof. Section 4 generalizes µdiv-simulation to a parameterized notion of simulation that captures 12 variants of strong and weak simulation, and studies this parameterized notion. Section 5 instantiates our parameterized simulation on the latest iteration of Choice Trees (CTrees), a data structure for representing the semantics of concurrent programs that notably lacked a notion of divergence-sensitive weak simulation until now, and establishes a parameterized proof system for simulation of CTrees. Section 6 instantiates µdiv-simulation on CompCert LTSs and demonstrates the applicability of our approach by porting a proof of correctness of an existing optimization pass to our framework. Section 7 discusses related work and in particular compares µdiv-simulation with two existing notions of simulation: FreeSim and weak-tau simulation.

2 Context

This section introduces important existing notions. It first dives into the world of simulations, starting from plain *strong simulation* (Section 2.1) then moving on to the most relevant notions of simulation for the present work: divergence-sensitive simulation (Section 2.2) and normed simulation (Section 2.3). Finally, it details the *coinduction up-to* proof approach in Section 2.4.

2.1 Labelled Transition Systems, Simulations

This section introduces the basic setting and terminology that this paper is based on: labelled transition systems and simulations.

The semantics of programs are usually modelled as *labelled transition systems* or LTSs. An LTS $(S, \Sigma, \rightarrow)$ consists of a (possibly infinite) set of states S, a (possibly infinite) set of labels Σ, and a labelled transition relation $\rightarrow \subseteq S \times \Sigma \times S$ [LTS.v:19]. We write $s \xrightarrow{l} s'$ for $(s, l, s') \in \rightarrow$.

In verified compilation, states of an LTS represent the possible states of execution of a program, and labelled transitions represent events that can occur during program execution. These LTSs are typically built from small-step operational semantics.

Simulation is the most commonly used form of behavioral inclusion in verified compilation, defined on LTSs. Its simplest form is *strong simulation.*

Definition 2.1. A relation $\mathcal{R}$ between states of an LTS[4] is a strong simulation if it verifies:

$$\forall t u, t \mathrel{\mathcal{R}} u \implies \texttt{ssimF}\ \mathcal{R}\ t\ u \text{ with } \texttt{ssimF}\ \mathcal{R}\ t\ u \triangleq \forall t'\ l, t \xrightarrow{l} t' \implies \exists u', u \xrightarrow{l} u' \wedge t' \mathrel{\mathcal{R}} u'$$

We say that u simulates t if $(t, u) \in \mathcal{R}$ for some simulation relation $\mathcal{R}$. In verified compilation, we aim to prove that the initial state of the program before compilation simulates the initial state of the program after compilation.

The `ssimF` function is called the *simulation game* or *simulation functor.* It can alternatively be characterized as a *simulation diagram*, as in Figure 2. In simulation diagrams, solid lines and arrows represent the universal quantifiers from the mathematical definition of the functor, and dashed ones represent the existential quantifiers.

[4]In practice, this may be a relation between states of two different LTSs, but the union of LTSs is a trivial operation [LTSSum.v:87].

Each transition from the left ("simulated") state t labelled with some l should be matched by a transition from u with the same label l, and the resulting pair of states should still be related by $\mathcal{R}$. Outgoing transitions from the left state t can be seen as *simulation challenges* to which the right state u has to answer.

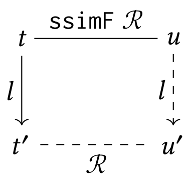

$$\begin{tikzcd} t \arrow[r, dash, "\texttt{ssimF}\ \mathcal{R}"] \arrow[d, "l"'] & u \arrow[d, dashed, "l"'] \\ t' \arrow[r, dashed, dash, "\mathcal{R}"'] & u' \end{tikzcd}$$

Fig. 2. The diagram characterizing a strong simulation relation $\mathcal{R}$.

The union of all strong simulations, noted `ssim`, is itself a strong simulation, dubbed *strong similarity*. As a consequence, the proposition `ssim` t u holds if and only if u simulates t. Similarity can be characterized as a greatest fixpoint: `ssim` $\triangleq$ `gfp ssimF`.

This section focused on the definition of strong simulation, but as we will soon see, more complex notions of simulation have the same ingredients: they relate states from two LTSs, they can be characterized by one or more simulation diagrams, and they induce a notion of similarity.

A note on undefined behaviors (UBs). In some verified compilers, including CompCert [22], simulations have built-in support for UBs: states in which "anything can happen" and that simulate every possible state. Languages such as C [20] feature UBs to increase optimization opportunities, though this design choice is subject to debate [31]. LTSs and simulations in the present paper do not feature built-in UBs. As a general rule, a UB state of execution can be encoded as a LTS state than can take transitions with every possible label[5] to every possible state [`UB.v:34`].

2.2 Divergence-sensitive Weak Simulation

Strong simulation is only the simplest of many existing notions of simulation. It is not typically used in verified compilation (except for simulation proofs based on big-step semantics, as in the early days of CompCert [7]) because it lacks a notion of transitions stemming from unobservable events. By contrast, *divergence-sensitive weak simulation* is a notion of behavioral inclusion of particular interest in the field of verified compilation. The main additional ingredient of *weak* simulations is that a particular label τ represents *internal* events that are unobservable. What precise kinds of event are represented as τ transitions or as observable transitions largely depends on design choices of individual semantics. Semantics in verified compilation usually represent as many events as possible as τ transitions, to allow for more optimizations. Indeed, if an event such as a memory access is observable, it means that the source program and the corresponding optimized compiled program have to perform exactly the same memory accesses in the same order, which greatly limits optimization opportunities. Observable events typically include external function calls and input/output operations.

Beyond treating τ transitions as invisible, most notions of simulation in verified compilation projects are *divergence-sensitive* [26, 44]. This condition ensures that if the simulated program admits an infinite series of invisible τ transitions, the simulating program admits one too.

Divergence-sensitive weak simulation can be defined diagrammatically, as in Figure 3. If the three diagrams hold for every pair of states in a relation $\mathcal{R}$, then $\mathcal{R}$ is a divergence-sensitive weak simulation. In these diagrams, $\xrightarrow{l^*}$ means "any finite number of successive transitions labelled with l". Likewise, $\xrightarrow{l^+}$ means "any non-null finite number of successive transitions labelled with l", and $\xrightarrow{l^\omega}$ means "an infinite number of successive transitions labelled with l". We note `divsimF` the simulation functor for divergence-sensitive weak simulation, and `divsim` the greatest divergence-sensitive weak simulation.

[5]Including an observable `UB` label representing behaviors not captured by existing labels but that can still be triggered by UBs.

$$\begin{tikzcd}
t \arrow[r, dash, "\texttt{divsimF}\ \mathcal{R}"] \arrow[d, "\tau"'] & u \arrow[d, dashed, "\tau^*"'] \\
t' \arrow[r, dash, dashed, "\mathcal{R}"'] & u'
\end{tikzcd}
\qquad
\begin{tikzcd}
t \arrow[r, dash, "\texttt{divsimF}\ \mathcal{R}"] \arrow[dd, "l"'] & u \arrow[d, dashed, "\tau^*"'] \\
 & {} \arrow[d, dashed, "l"'] \\
t' \arrow[r, dash, dashed, "\mathcal{R}"'] & u'
\end{tikzcd}
\qquad
\begin{tikzcd}
t \arrow[r, dash, "\texttt{divsimF}\ \mathcal{R}"] \arrow[d, "\tau^\omega"'] & u \arrow[d, dashed, "\tau^\omega"] \\
{} & {}
\end{tikzcd}$$

Fig. 3. The three diagrams that classically characterize a divergence-sensitive weak simulation relation $\mathcal{R}$.

The two diagrams on the left of Figure 3 define a weak simulation. Each transition from the left ("simulated") state t labelled with $l \neq \tau$ should be matched by a transition with the same label, with the possible addition of a finite number of τ transitions before.[6] Each transition from t labelled with τ should be matched by a finite number of τ transitions. These two cases capture that τ transitions are semantically "invisible" and can be skipped when playing the simulation game.

The third case captures sensitivity to silent divergence: when t can perform an infinity of τ transitions, u should be able to perform an infinity of τ transitions too. Without this third diagram, a state s that loops on itself with a τ transition is simulated by *any* state s'. Indeed, the relation $\mathcal{R} = \{(s, s')\}$ is a weak simulation: whenever s loops on itself, s' can stagnate in response, as is permitted by the τ diagram of weak simulation, and the resulting pair (s, s') is in $\mathcal{R}$ despite no progress having been made. This condition is important for verified compilation. Recall that the correctness of a verified compilation pass relies on a proof that the compiled program $\mathcal{C}(P)$ is simulated by the source program P. The program `while (true) {}` is typically modelled as a state that τ-loops on itself. Transporting the above result of weak simulation between s and s', it is correct for weak simulation to compile any program to `while (true) {}`. Of course, this is not a desirable property, which is why verified compilers rely on variants of divergence-sensitive simulation. More generally, divergence sensitivity ensures that if the compiled program silently diverges, the source program diverges too.

There is a fundamental difference between the divergence sensitivity diagram and the two previous diagrams: the two diagrams for weak simulation model a case analysis on the transitions that are immediately possible from t, whereas the diagram for divergence preservation quantifies on *infinite sequences of future τ transitions.* This makes this divergence preservation condition *global.* A proof scientist willing to prove a divergence-sensitive weak simulation result using this basic definition would have proof obligations that involve reasoning globally about the future of the LTS. In practice, this is too much of a burden and frameworks related to verified compilation do *not* use this definition, but variants devised to alleviate the need for such a global condition.

2.3 Normed Simulation

Notice that of the two cases of the classical weak simulation diagram from Figure 3, the only case that is problematic with respect to divergence preservation is when t perfoms a τ transition and u stagnates with a τ^0 answer. Indeed, if u answers with at least one τ, it means that u performs at least as many τ transitions as t, which ensures that whenever t diverges, u diverges too. This observation is at the basis of variants of divergence-sensitive weak simulation without a global condition. In particular, normed simulation [18] has emerged in the 90s and is now widely used in verified compilation projects, as well as in model checking [4].

[6]Alternatively, the weak simulation game for $l \neq \tau$ can be defined with optional τ^* before *and after* the answer labelled with l. These variants are known to be equivalent.

$$\begin{tikzcd} t \arrow[r, dash, "\texttt{nsimF}\ \mathcal{R}"] \arrow[d, "\tau"'] & u \arrow[d, dashed, "\tau^+"'] \\ t' \arrow[r, dash, dashed, "\mathcal{R}"'] & u' \end{tikzcd} \qquad \begin{tikzcd} t \arrow[r, dash, "\texttt{nsimF}\ \mathcal{R}"] \arrow[dd, "l"'] & u \arrow[d, dashed, "\tau^*"'] \\ & {} \arrow[d, dashed, "l"'] \\ t' \arrow[r, dash, dashed, "\mathcal{R}"'] & u' \end{tikzcd} \qquad \begin{tikzcd} t \arrow[r, dash, "\texttt{nsimF}\ \mathcal{R}"] \arrow[d, "\tau"'] & u \arrow[dl, dash, dashed, "\mathcal{R} \wedge m(t') < m(t)"] \\ t' & \end{tikzcd}$$

Fig. 4. The three diagrams that characterize an explicit normed simulation relation $\mathcal{R}$.

Normed simulation is a more restrictive form of divergence-sensitive weak simulation that relies on a notion of well-founded measure m that can be expressed locally instead of a global condition. We will call it explicit normed simulation in the following, in order to distinguish it from the notion of simulation described at the end of the present section. It is also known as explicit stuttering simulation [15]. Its definition is given diagrammatically in Figure 4, noting `nsimF` its functor.

This time, two diagrams have the same challenge $\xrightarrow{\tau}$. This means that when there is a τ transition, u can answer with either of these diagrams (this convention will apply throughout this paper). This split is critical to ensure divergence preservation. Indeed, this notion of simulation is parameterized with a well-founded measure m that has to decrease when t perfoms a τ transition and u stagnates in response. This means that this specific stagnation diagram can only be invoked a *finite* number of consecutive times before the measure reaches zero and the diagram becomes invalid. In other words, t can only perform a finite number of unanswered τ transitions before u starts moving too.

CompCert has been successfully using variants of explicit normed simulation for a long time [22]. When working with deterministic LTSs (LTSs in which outgoing transitions of a given state are all labelled differently), explicit normed simulation is complete with respect to divergence-sensitive weak simulation. This is the case for most of the CompCert compilation chain, but in other settings (especially concurrent settings) deterministic LTSs are often insufficiently expressive, and explicit normed simulation is not complete with respect to divergence-sensitive weak simulation for nondeterministic LTSs, which limits the applicability of normed simulation. A second limitation is that having to define a well-founded measure on states of the LTS can be uncomfortable. There have been efforts to address the discomfort of the well-founded measure on states, as is detailed in the next section.

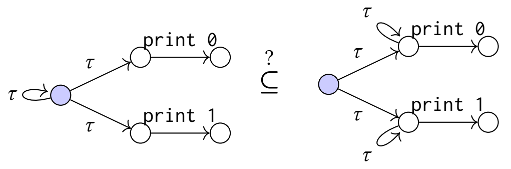
However, we are not aware of any notion of simulation that addresses the incompleteness of explicit normed simulation in the general case, while retaining the possibility of local reasoning. Going back to the example in Figure 1, attempts to prove that the behaviors of t_0 are included in the behaviors of u_0 using explicit normed simulation are doomed to fail. Suppose there exists an explicit normed simulation relation $\mathcal{R}$ that contains (t_0, u_0), and an associated measure m. The $t_0 \xrightarrow{\tau} t_0$ transition has to be simulated in some way by u_0. Following the simulation diagrams from Figure 4, u_0 can either perform a series of τ transitions, or stay in place provided the measure m associated with $\mathcal{R}$ decreases.

- The answer $u_0 \xrightarrow{\tau} u_1$ is not possible, because if t_0 subsequently moves to t_3 and then t_4, u_1 has no way to simulate a transition labelled with b.
- The alternative is for u_0 not to move, but in this case the measure has to strictly decrease. It is clear that $m(t_0) < m(t_0)$ cannot hold for any measure based on a well-founded order.

Thus u_0 does not simulate t_0 for explicit normed simulation. This incompleteness of normed simulation is a well-known fact in the model checking community [4].

More specifically, divergence-sensitive weak simulation supports *commutation between nondeterministic τ transitions and possible divergence*, while normed simulation does not. For a more

```
int x;
while(rand() % 2) {}
x = rand() % 2;
printf("%d", x);
```



```
int x;
x = rand() % 2;
while(rand() % 2) {}
printf("%d", x);
```

Fig. 5. Two C programs and associated LTSs. Normed simulation cannot capture this refinement result.

concrete illustration, consider the two C programs depicted in Figure 5. Corresponding (slightly simplified) LTSs are depicted alongside them, assuming `printf` is modelled as a visible event, and `rand() % 2` as two τ transitions. Because the `rand` calls are modelled as nondeterministic τ transitions, and the `while` loops are modelled as possibly diverging states with τ loops, the desired simulation result cannot be proved using normed simulation (for reasons similar to the previously discussed example), only with divergence-sensitive weak simulation. Decoupling the handling of silent divergence from the handling of τ transitions is a key feature of the notion of simulation we introduce in Section 3. Beyond artificial `rand` loops, synchronization primitives in a concurrent setting may generate this kind of LTS with possible silent divergence.

Implicit normed simulation. Implicit normed simulation [29] is a more modern take on normed simulation that makes use of a mixed inductive-coinductive definition to circumvent the need for a decreasing measure. The notion of simulation is defined coinductively, but the case in which u stagnates is defined *inductively*, ensuring that it can only be invoked a finite number of consecutive times. It is equivalent to explicit normed simulation: the simulation example from Figure 1 cannot be proved with implicit normed simulation either. The simulation diagrams of implicit normed simulation are the same as explicit normed simulation (see Figure 4), except that the conclusion of the rightmost diagram is replaced by `isimF` $\mathcal{R}$ (noting `isimF` the functor for implicit normed simulation). This makes `isimF` an inductive definition: the (t', u) pair resulting from the application of this diagram shall in turn be related by one of the three diagrams. In practice, `isim` is defined as the greatest fixpoint *of a least fixpoint*, and the third diagram is defined inductively rather than coinductively. We omit the details for implicit normed simulation, but the formal definition of a similar inductive-coinductive relation is given in Section 3.1.

These past few years, implicit normed simulation has been frequently favored in programming language theory: it is in particular the basis of the `sutt` simulation and the `eutt` bisimulation of ITrees [47], as well as Simuliris [16]. But this more refined notion suffers from the same flaw as explicit normed simulation: it is more restrictive than divergence-sensitive weak simulation.

2.4 Coinduction up-to

The most straightforward way to prove a refinement result between two LTS states t and u is to exhibit a relation $\mathcal{R}$ such that $t \mathcal{R} u$ and prove that this $\mathcal{R}$ is a simulation. A major drawback is that this approach requires to fully exhibit this possibly complex relation $\mathcal{R}$ from the beginning. In CompCert, this relation is typically called `match_states`, and for some compilation passes its definition can take several dozen lines of code. In a proof assistant such as Rocq, it is more comfortable to start from an incomplete relation and interactively refine it bit by bit. This is precisely what modern *coinduction up-to* permits. This section briefly sums up the possibilities offered by coinduction up-to for proofs of simulation, with in particular a proof example.

Coinduction up-to has a long history, dating back to the 80s [35], and underlying theories have been developing rapidly in the 2010s. In particular, the advent of *parameterized coinduction* [19] and then the more general *coinduction up-to companion* method [33, 38] marked a turning point for

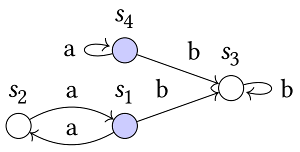


Fig. 6. An example LTS in which s_4 strongly simulates s_1

interactive coinductive proofs in Rocq. A major motivation for the development of coinduction up-to techniques was bisimulations [34], but these results have not necessarily been extended to the simpler setting of simulations. Nevertheless, Interaction Trees [47] and then Choice Trees [13] are recent Rocq libraries that define notions of simulation for verified compilation based on coinduction up-to.

With coinduction up-to, the goal of a simulation proof is no longer to prove that a given relation is a simulation, but that it is a subrelation of *similarity*, the simulation defined as the union of all simulations. For that purpose, a plethora of proof techniques dubbed *up-to techniques* can be used. They are generally of the shape "*if these pairs of states are contained in the similarity relation, so are these other pairs of states*". The initial relation is called the *simulation candidate*. When conducting a proof by coinduction up-to, one does not work directly on the candidate relation $\mathcal{R}$ but on the candidate relation *up-to companion*: the relation $t_{\texttt{ssimF}}(\mathcal{R})$, noted $`\mathcal{R}$ for brevity in the rest of this paper. The function $t_{\texttt{ssimF}}$ is the union of all compatible up-to techniques for the (in this case, strong) simulation functor, itself a compatible up-to technique dubbed the *companion*n The crucial consequence of working with $`\mathcal{R}$ is that compatible up-to techniques can be invoked on the fly during a proof of simulation. We omit the theory behind it, as it would be too technical and it is nicely encapsulated in modern Rocq coinduction libraries.

With coinduction up-to, it is no longer necessary to exhibit a simulation relation from the beginning of a proof, one can start from a smaller relation and refine it as needed using up-to techniques. To give an idea of how proofs by coinduction up-to are structured, we give below two alternative proofs that s_4 strongly simulates s_1 in the LTS depicted in Figure 6: a classical proof, and another one by coinduction up-to.

Classical proof. We take $\mathcal{R} = \{(s_1, s_4), (s_2, s_4), (s_3, s_3)\}$. For each of these pairs, we now check that the strong simulation diagram holds: $\forall (s, s') \in \mathcal{R}, \texttt{ssimF}\ \mathcal{R}\ s\ s'$. For instance, starting from (s_1, s_4), there are two transitions that s_1 can take:

- $s_1 \xrightarrow{a} s_2$. In this case, s_4 can answer with $s_4 \xrightarrow{a} s_4$ and we indeed have $(s_2, s_4) \in \mathcal{R}$.
- $s_1 \xrightarrow{b} s_3$. In this case, s_4 can answer with $s_4 \xrightarrow{b} s_3$ and we indeed have $(s_3, s_3) \in \mathcal{R}$.

We omit similar checks for the pairs (s_2, s_4) and (s_3, s_3). As the strong simulation diagram holds for $\mathcal{R}$, this relation is a strong simulation that contains in particular (s_1, s_4), which concludes the proof.

Proof by coinduction up-to. With coinduction up-to, we do not have to exhibit a simulation relation $\mathcal{R}$ from the start. We can instead plainly start from $\mathcal{R} = \{(s_1, s_4)\}$ and refine this *simulation candidate* as the proof progresses.

In particular, three standard simulation up-to techniques will be useful for this proof:

- The simulation up-to simulation step technique, stating that $\forall s t, \texttt{ssimF}\ `\mathcal{R}\ s\ t \implies s\ `\mathcal{R}\ t$.
- The up-to reflexivity technique, stating that $\forall s, s\ `\mathcal{R}\ s$.
- The up-to identity technique, stating that $\forall s t, s\ \mathcal{R}\ t \implies s\ `\mathcal{R}\ t$.

Again, we have to prove that the simulation diagram holds, but this time the proof goal involves the simulation candidate up-to companion: $\forall (s, s') \in \mathcal{R}$, `ssimF` $`\mathcal{R}\ s\ s'$. After unfolding $\mathcal{R}$, this can be reformulated as the following initial proof state: $\vdash$ `ssimF` $`\mathcal{R}\ s_1\ s_4$.

In an initial proof state, the goal always involves `ssimF` $`\mathcal{R}$ and not $`\mathcal{R}$. Consequently the coinduction hypothesis cannot be applied (otherwise all coinductive proofs would be immediate!), we say that it is *guarded*. Unlike Rocq's native tools for coinduction, this guard is not syntactic but semantic.

First, we unfold the simulation game. There are two possible transitions from s_1.

- $s_1 \xrightarrow{a} s_2$. In this case, s_4 can answer with $s_4 \xrightarrow{a} s_4$. The new proof goal is: $\vdash\ s_2\ `\mathcal{R}\ s_4$. The relation $`\mathcal{R}$ is now exposed, the goal is *unguarded* and the coinduction hypothesis could be applied if it matched. In this case it doesn't: the coinduction hypothesis only contains (s_1, s_4) but not the pair of interest (s_2, s_4). To get back to the coinduction hypothesis, we apply the *simulation up-to simulation step* technique: $\vdash$ `ssimF` $`\mathcal{R}\ s_2\ s_4$.
  We play the simulation game: $s_2 \xrightarrow{a} s_1$ is answered by $s_4 \xrightarrow{a} s_4$: $\vdash\ s_1\ `\mathcal{R}\ s_4$. We got back to (s_1, s_4), the up-to identity technique concludes this first case.
- $s_1 \xrightarrow{b} s_3$. In this case, s_4 can answer with $s_4 \xrightarrow{b} s_3$. The new proof goal is: $\vdash\ s_3\ `\mathcal{R}\ s_3$. We apply the *up-to reflexivity* technique, which concludes this second case.

In this second proof, we did not have to explicitly add the intermediate pair (s_2, s_4) to the simulation relation. The pair (s_2, s_4) was already "lazily" in $\mathcal{R}$ *up-to one simulation step*, and thus in $\mathcal{R}$ up-to the union of all compatible up-to principles, i.e. in $`\mathcal{R}$.

As a final note, a convenient result about coinduction up-to states that any property valid for $`\mathcal{R}$ are also valid for `ssimF` $`\mathcal{R}$ and `ssim`. Of course, this result is stated for `ssimF` here but it is just as valid for any other functor.

In the rest of this paper, we will always implicitly work with simulation candidates up-to companion, thus we will omit the backticks and write e.g. $\mathcal{R}$ for $`\mathcal{R}$.

3 A Mutually Coinductive Characterization of Divergence-Sensitive Weak Simulation

Divergence-sensitive weak simulation from the previous century was not particularly usable in practice, which led to the definition of the previously mentioned notions of simulation.

However, the state of the art regarding coinduction has significantly evolved since then, in particular coinduction up-to allows to define particularly involved proof techniques on coinductive relations such as (bi)simulation. To date, the verified compilation community has not fully adopted this advance that originates from the process algebra community. The most extensive use of coinduction up-to in a project adjacent to verified compilation can arguably be found in the Choice Trees library [13], but divergence-sensitive weak simulation is not among the notions of (bi)simulation it defines. Defining a modern notion of divergence-sensitive weak simulation thus remains an open problem. This section tackles it with two design principles:

- This new simulation should be sound and complete with respect to the original divergence-sensitive weak simulation.
- The definition of this new simulation should be as simple as possible, and advanced reasoning techniques can be recovered through up-to techniques.

Additionally, one key observation guided the design of this novel notion of simulation: as discussed in Section 2.2, the problem with divergence-sensitive weak simulation is its global characterization of divergence preservation, which is why the next section will build a *local* characterization of divergence preservation.

$$\begin{tikzcd}
t \arrow[r, dash, "\texttt{divpresIndF}\ \mathcal{R}_{div}"] \arrow[d, "\tau"'] & u \arrow[d, dashed, "\tau"'] \\
t' \arrow[r, dash, dashed, "\mathcal{R}_{div}"'] & u'
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
t \arrow[r, dash, "\texttt{divpresIndF}\ \mathcal{R}_{div}"] \arrow[d, "\tau"'] & u \\
t' \arrow[ur, dash, dashed, "\mathcal{R}^{ind}_{div}"'] &
\end{tikzcd}$$

Fig. 7. An inductive-coinductive characterization of divergence preservation, based on a functor `divpresIndF`

3.1 Divergence Preservation, Coinductively

The divergence preservation diagram from Figure 3 is a global property. In this section, we propose an alternative inductive-coinductive definition formulated like a simulation game. It is depicted in Figure 7.

By itself, the left diagram ensures that if t diverges, then u diverges too. As for the right diagram, it comes useful in case neither t nor u can diverge. For instance, if there is one (or a finite number of) τ transition at t but zero at u, neither t nor u diverge, thus divergence is preserved, but the left diagram does not capture it, only the right one does. As an *inductive* rule, the right diagram does not consume the coinductive guard: the inductive-coinductive definition will indeed be set up so that $\mathcal{R}^{ind}_{div} \iff \texttt{divpresIndF}\ \mathcal{R}_{div}$.

Formally, we define the functor corresponding to these diagrams as

$$\texttt{divpresIndF}\ \mathcal{R}_{div}\ \mathcal{R}^{ind}_{div}\ t\ u \triangleq \forall t',\ t \xrightarrow{\tau} t' \implies (\exists u',\ u \xrightarrow{\tau} u' \land \mathcal{R}_{div}\ t'\ u') \lor (\mathcal{R}^{ind}_{div}\ t'\ u).$$

Then, we apply a least fixpoint operator (through Rocq's built-in inductive type mechanism) and a greatest fixpoint operator (provided by the `coinduction` library) [Divergence.v:207]:

$$\texttt{divpres} \triangleq \texttt{gfp}\ \texttt{divpresF}\ \text{with}\ \texttt{divpresF}\ \mathcal{R}_{div} \triangleq \texttt{lfp}\ \left(\lambda \mathcal{R}^{ind}_{div} \cdot \texttt{divpresIndF}\ \mathcal{R}_{div}\ \mathcal{R}^{ind}_{div}\right)$$

The resulting relation captures divergence preservation, and it does so locally: unlike the divergence preservation condition from Figure 3, the challenge of this coinductive notion of divergence preservation does not involve τ^{ω} but *just one* τ transition.

Theorem 3.1 (`divpres` is equivalent to divergence preservation [Divergence.v:322]). *For all states t and u,* `divpres` $t\ u$ *implies* `diverges` $t \implies$ `diverges` u. *In classical logic, the converse is also true.*

In this definition of divergence preservation, it can be inconvenient to have to match each τ from t with *exactly* one τ from u. Fortunately, an up-to technique can relax this condition to *one or more* τ transitions.

Lemma 3.2 (Divergence preservation up-to τ [Divergence.v:257]). *If $\mathcal{R}_{div}$ is a candidate for* `divpresF`*, then*

$$\left(\forall t',\ t \xrightarrow{\tau} t' \implies (\exists u',\ u \xrightarrow{\tau^+} u' \land \mathcal{R}_{div}\ t'\ u') \lor (\mathcal{R}^{ind}_{div}\ t'\ u)\right) \implies \texttt{divpresIndF}\ \mathcal{R}^{ind}_{div}\ \mathcal{R}_{div}\ t\ u.$$

3.2 Definition of μdiv-simulation

We propose a novel notion of simulation based on definitions of simulation and divergence preservation stated in a mutually coinductive manner.

Again, we give a diagrammatic definition in Figure 8. Just like normed simulation, the τ case is split into two so as to ensure divergence preservation in the case where u stagnates. By contrast, deadlock preservation is not ensured by this definition, see Section 4 for such a generalization.

$$\begin{tikzcd}
t \arrow[r, dash, "\mathtt{\mu simF}\ \mathcal{R}"] \arrow[d, "\tau"'] & u \arrow[d, dashed, "\tau^+"'] \\
t' \arrow[r, dash, dashed, "\mathcal{R}"'] & u'
\end{tikzcd}
\qquad
\begin{tikzcd}
t \arrow[r, dash, "\mathtt{\mu simF}\ \mathcal{R}"] \arrow[dd, "l"'] & u \arrow[d, dashed, "\tau^*"'] \\
 & {} \arrow[d, dashed, "l"'] \\
t' \arrow[r, dash, dashed, "\mathcal{R}"'] & u'
\end{tikzcd}
\qquad
\begin{tikzcd}
t \arrow[r, dash, "\mathtt{\mu simF}\ \mathcal{R}"] \arrow[d, "\tau"'] & u \\
t' \arrow[ur, dash, dashed, "\mathcal{R} \cap \mathcal{R}_{div}"'] &
\end{tikzcd}$$

Fig. 8. The definition of µdiv-simulation

This time, the technique used to ensure the preservation of divergence is *mutual coinduction*: in the third diagram of $\mathcal{R}$ the game not only leads back to $\mathcal{R}$ but also to $\mathcal{R}_{div}$, a divergence preservation candidate as defined in the previous section. In practice, this means that the simulation functor does not operate on a pair of states but on a triple made of a pair of states and a boolean that indicates whether the two current states are related by $\mathcal{R}$ or $\mathcal{R}_{div}$.[7] Formally, the `µsimF` functor is defined as follows:

$$\mathtt{\mu simF}\ \mathcal{R}\ \mathcal{R}_{div}\ t\ u \triangleq \left(\forall l t',\ l \neq \tau,\ t \xrightarrow{\tau^*}\xrightarrow{l} t' \implies \exists u',\ u \xrightarrow{l} u' \land t'\ \mathcal{R}\ u'\right) \land$$
$$\left(\forall t',\ t \xrightarrow{\tau} t' \implies ((\exists u', u \xrightarrow{\tau^+} u' \land t'\ \mathcal{R}\ u') \lor (t'\ \mathcal{R}\ u \land t'\ \mathcal{R}_{div}\ u))\right)$$

It may not be clear at first sight how this definition is more comfortable than the one with a global divergence preservation condition. This definition gets rid of the global condition, but the apparent need to work with two relations $\mathcal{R}$ and $\mathcal{R}_{div}$ can be intimidating. Fortunately, the $\mathcal{R}_{div}$ relation can be hidden in higher-level proof principles. The one case of the simulation game in which $\mathcal{R}_{div}$ appears is the one where a τ transition is consumed on the t side, but not on the u side. A natural way to prove $\mathcal{R}_{div}$ is to prove a global divergence preservation result in the style of divergence-sensitive weak simulation.

Lemma 3.3 (Link between $\mathcal{R}_{div}$ and `divpres` [Sims.v:408]). *Let $(\mathcal{R}, \mathcal{R}_{div})$ be a µdiv-simulation candidate. For all states t and u,* `divpres` $t\ u \implies \mathcal{R}_{div}\ t\ u$.

As discussed in Section 2.2, such a global property can be uncomfortable. It would be very valuable to have a proof technique that allows consuming such a τ transition on only the t side, without having to work with $\mathcal{R}_{div}$. In implicit normed simulation-based developments, this property is typically encoded directly into the simulation functor, and the asynchronous consumption of a τ is *inductive* rather than being coinductive. This is not the case in our definition of simulation, but we can resort to a coinduction up-to technique to enable this kind of reasoning.

3.3 Up-to tau Techniques

Simulation techniques that allow asymmetric consumption of τ transitions give proof scientists the opportunity to write more compositional proofs.

The left up-to τ technique allows consuming τ transitions from the left state without the $\mathcal{R}_{div}$ side condition.

Lemma 3.4 (Link between $\mathcal{R}$ and $\mathcal{R}_{div}$ [Sims.v:394]). *Let $(\mathcal{R}, \mathcal{R}_{div})$ be a µdiv-simulation candidate. For all states t and u,* `µsimF` $\mathcal{R}\ t\ u \implies \mathcal{R}_{div}\ t\ u$.

[7] When referring to $\mathcal{R}_{div}$, we implicitly refer to the $\mathcal{R}_{div}$ mutually defined with $\mathcal{R}$.

Whenever we have to prove that two states are related by $\mathcal{R}_{div}$, we can prove that they are related by `µsimF` $\mathcal{R}$ instead, which allows one to ignore $\mathcal{R}_{div}$ and only work with $\mathcal{R}$ in proofs of simulation. As an immediate consequence, the left up-to τ diagram from Figure 9 can be used instead of the $\mathcal{R}_{div}$ case in the µdiv-simulation game from Figure 8.

Similarly, a right up-to τ technique is stated below: we can always choose a τ transition on the u side and consume it.

$$\begin{tikzcd}
t \arrow[r, dash, "\texttt{µsimF}\ \mathcal{R}"] \arrow[d, "\tau"'] & u \\
t' \arrow[ur, dashed, dash, "\texttt{µsimF}\ \mathcal{R}"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}"] \arrow[dr, dashed, dash, "\mathcal{R}"'] & u \arrow[d, dashed, "\tau"] \\
& u'
\end{tikzcd}$$

Fig. 9. The left up-to τ technique [Sims.v:941] and the right up-to τ technique [Sims.v:453]

Lemma 3.5 (Right up-to τ technique [Sims.v:453]). *Let $(\mathcal{R}, \mathcal{R}_{div})$ be a µdiv-simulation candidate.* $\forall t\, u\, u', u \xrightarrow{\tau} u' \wedge t\ \mathcal{R}\ u' \implies t\ \mathcal{R}\ u$.

This technique acts on $\mathcal{R}$ but an up-to technique valid for $\mathcal{R}$ is also valid for `µsimF` $\mathcal{R}$: it can be used both on a guarded or unguarded goal.

We believe this up-to technique is a great illustration of the power of coinduction up-to. In ITrees and FreeSim, the simulation functor does not consist of standard simulation diagrams but of a hardcoded sound and complete proof system that includes in particular these two proof rules.[8] Here, we prefer to define a more direct definition of `µsimF` and recover more involved proof principles through coinduction up-to.

3.4 Comparison with Divergence-Sensitive Weak Simulation

A major difference between µdiv-simulation and divergence-sensitive weak simulation is that the $\mathcal{R}_{div}$ in the simulation diagram represents a partially-built proof of divergence preservation, whereas the definition of divergence-sensitive weak simulation relies on a global divergence preservation condition, which is naturally stronger. This means in particular that it is always possible to exhibit a global proof of divergence preservation instead of proving that two states are in $\mathcal{R}_{div}$, using Lemma 3.3. Proofs using this property will be very close to proofs based on standard divergence-sensitive weak simulation, but a strength of our definition is that it is possible to use both this method and the left up-to τ diagram at different points of the same proof of simulation. This is precisely what we do in Section 3.5.

Ultimately, `µsim` and `divsim` define the same coinductive relation.

Theorem 3.6 (Equivalence with standard divergence-sensitive weak simulation [DivSim.v:87]). $\forall t\, u$, `µsim` $t\, u \implies$ `divsim` $t\, u$. *Moreover, in classical logic, the converse is also true.*

This result raises the question of why we did not just use the standard definition of `divsim`, and recover the desired proof techniques (e.g., the up-to τ techniques) through coinduction up-to. The reason is that an up-to technique valid on some characterization of a coinductive relation is not necessarily valid on other characterizations of the same coinductive relation. In this specific case, we could not prove the important left up-to τ technique for `divsim`.

This result of equivalence with divergence-sensitive weak simulation is uncommon for notions of simulation used in verified compilation, as most of them are equivalent to normed simulation (with the notable exception of weak-tau simulation, as studied in Section 7), which is sound but not complete with respect to divergence-sensitive weak simulation. The mutually coinductive characterization of divergence preservation was key to achieve that.

[8]Amusingly, the `sutt` relation on ITrees stands for *simulation up-to tau*, and can be seen as an implicit normed simulation with a hardcoded right up-to τ technique.

As a final note, we also defined implicit normed simulation in Rocq, proved that it implies µdiv-simulation[IndSim.v:101] and proved that on a simple LTS µdiv-simulation holds but not implicit normed simulation, confirming that normed simulation is more restrictive [IndSimCounterExample.v:88].

3.5 A Proof of Simulation

This section demonstrates how to establish the simulation result between the two LTSs shown in the introduction (Figure 1). We also proved it in Rocq [SimExample.v:55].

We would like to show `µsim` t_0 u_0 using µdiv-simulation and up-to techniques, starting from a simple simulation candidate $\mathcal{R}$ containing only (t_0, u_0). The initial proof goal is: $\vdash$ `µsimF` $\mathcal{R}$ t_0 u_0.

Unfolding the simulation game, there are three possible transitions from t_0 that should be taken into account.

- $t_0 \xrightarrow{\tau} t_0$ can be answered by u_0 stagnating. We can try to do so using the left up-to τ technique, which yields: $\vdash$ `µsimF` $\mathcal{R}$ t_0 u_0.
  We got back our previous proof goal, so the up-to τ technique is not a viable approach here. We have to use the more general stagnation diagram from Figure 8 with Lemma 3.3, which yields: $\vdash\ t_0\ \mathcal{R}\ u_0 \wedge t_0\ \mathcal{R}_{div}\ u_0$.
  The $\mathcal{R}$ part of the proof goal is immediate from the coinduction hypothesis. As for the $\mathcal{R}_{div}$ goal, one can prove `divpres` t_0 u_0. This reduces to `divpres` t_0 u_1, which is easily proved as u_1 is a silently diverging state [SimExample.v:44].
- $t_0 \xrightarrow{\tau} t_1$ can be answered by $u_0 \xrightarrow{\tau} u_1$, yielding: $\vdash\ t_1\ \mathcal{R}\ u_1$.
  We apply the up-to simulation step technique: $\vdash$ `µsimF` $\mathcal{R}$ t_1 u_1.
  We unfold the simulation game. To $t_1 \xrightarrow{a} t_2$, u_1 answers $u_1 \xrightarrow{a} u_2$. The proof goal becomes: $\vdash\ t_2\ \mathcal{R}\ u_2$.
  Again, we apply the up-to simulation step technique, and play the simulation game. As there is no outgoing transition from t_2, this concludes this case.
- $t_0 \xrightarrow{b} t_3$ can be answered by $u_0 \xrightarrow{b} u_3$, yielding: $\vdash\ t_3\ \mathcal{R}\ u_3$.
  We apply the up-to simulation step technique. The only outgoing transition from t_3 is $t_3 \xrightarrow{\tau} t_4$, to which u_3 can only answer by stagnating. This time it will do so using the left up-to τ technique, so that we do not have to perform another proof of divergence preservation. The goal becomes: $\vdash$ `µsimF` $\mathcal{R}$ t_4 u_3
  As before, there is no outgoing transition from t_4, so we can unfold the simulation game to conclude this final case.

This result relies on a divergence preservation proof (in its first case), a kind of reasoning that is not available with variants of normed simulation, which explains why our proof attempt in Section 2.3 failed.

4 A Parameterized Notion of Simulation

The previous section introduced µdiv-simulation, a novel characterization of divergence-sensitive weak simulation, and defined a few basic reasoning techniques. The next step would naturally be to prove more results about this simulation. But before getting to that, this section will first generalize the previously introduced µdiv-simulation to a parameterized notion of simulation that captures not only divergence-sensitive weak simulation but also strong simulation, weak simulation, deadlock-sensitive simulation, and more.

The point is that there exist many subtle variants of simulation, whose associated theories are vastly similar. Studying them jointly allows to factor out a lot of work, and is particularly relevant

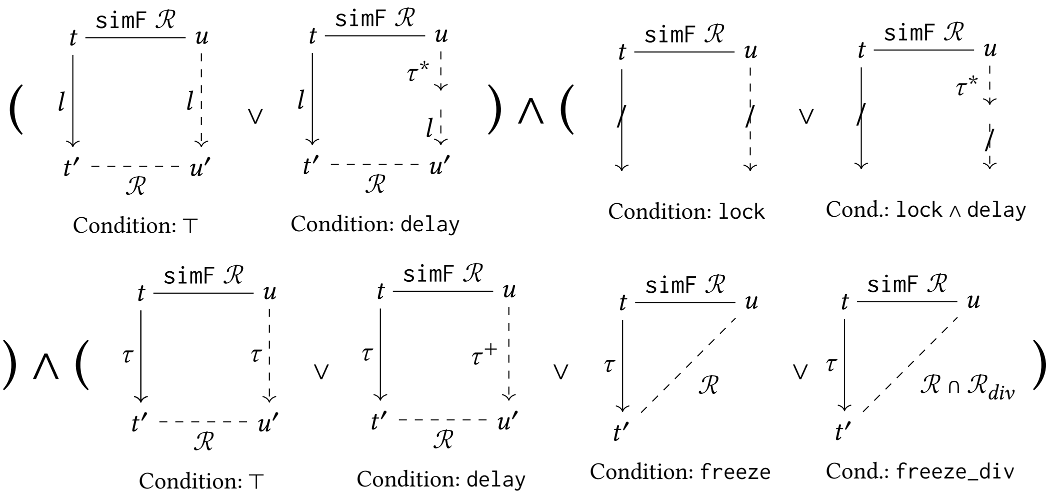


Fig. 10. The parameterized definition of μdiv-simulation[Sims.v:124]. Definition of the $\mathcal{R}_{div}$ game omitted (see Figure 7). In the nolock case, the second disjunction can be ignored.

when many variants of simulation have, to our knowledge, never been studied through the lens of coinduction up-to.

This section considers three parameters that effect the simulation game in distinct ways, and that can be combined to yield various notions of simulation, with a total of 12 possible combinations.

4.1 Definition

The parameterized simulation functor `simF` is indexed by three parameters that tweak the simulation game:

- $\mathcal{D} \in \{$`delay`, `nodelay`$\}$: When t performs a transition, should u answer immediately with the same transition (`nodelay`) or can u delay its answer with a series of τ transitions (`delay`)?
- $\mathcal{F} \in \{$`freeze`, `freeze_div`, `nofreeze`$\}$: When t performs a τ transition, can u stay in place (`freeze`) or does it necessarily have to answer with a τ transition (`nofreeze`)? If it can stay in place, should we make sure divergence is preserved (`freeze_div`)?
- $\mathcal{L} \in \{$`nolock`, `lock`$\}$: When t is stuck and cannot take any transition, should u be required to be stuck as well (`lock`)?

The parameterized simulation diagrams implementing this concept can be found in Figure 10. In this figure, $t \not\rightarrow$ means that t has no outgoing transition and is thus stuck. The disjunctions in the figure materialize different possible ways of answering a same simulation challenge. Some of the diagrams are conditioned on the value of the parameters, they are omitted if the condition does not match.

The non-parameterized μdiv-simulation studied in the previous section corresponds to $\mathcal{D}$ = `delay`, $\mathcal{F}$ = `freeze_div`, and $\mathcal{L}$ = `nolock`. Other combinations of the parameters lead to distinct notions of simulation, some of them being well-known. Table 1 sums up the possible combinations of $\mathcal{D}$ and $\mathcal{F}$, assuming $\mathcal{L}$ = `nolock`. Strong simulation, weak simulation, and divergence-sensitive simulation are standard. The three other notions are less interesting by themselves, but are valuable proof devices for variants of weak simulation, as Section 4.3 will establish. The name "plus simulation" comes from the fact that for this notion the answer to a τ challenge has the shape τ^+. This name has also been used with a similar definition in CompCert [22]. Analogously, the name

Table 1. Notions of simulation we obtain depending on the parameters `freeze` and `delay`

$\mathcal{D}$ / $\mathcal{F}$	`nofreeze`	`freeze_div`	`freeze`
`nodelay`	strong	divergence-sensitive "option"	"option"
`delay`	"plus"	divergence-sensitive weak	weak

"option simulation" comes from the fact that for this notion the answer to a τ challenge has to be either a τ transition, or nothing. Option simulation is reminiscent of the expansion preorder used in a weak *bi*simulation setting [1], and known as `euttge` in the Interaction Trees library.

The third dimension, $\mathcal{L}$, is not depicted in Table 1. When $\mathcal{L}$ = `lock`, one can simply prepend "*deadlock-sensitive*" or "*complete*" to the names in Table 1. Indeed, it materializes *deadlock sensitivity*. The story of deadlock sensitivity is similar to the story of divergence sensitivity from Section 2.2: with a standard weak simulation, the empty process (a stuck state that cannot take any transition) is trivially simulated by every possible state, and as a result compiling any program to a stuck program is correct with respect to weak simulation. Deadlock sensitivity is however not as critical as divergence sensitivity because LTSs for programming language semantics can often be built to not have any stuck state, making deadlock sensitivity irrelevant. When deadlock sensitivity is still needed, it is relatively straightforward to augment a notion of simulation with it, as does our parameter $\mathcal{L}$. Note that beyond complete simulations, there are other notions of deadlock-sensitive simulations, such as ready simulation [9, 36], which is out of the scope of the present paper. Which one to choose depends on the use case: complete simulation is generally appropriate for the formalization of programming languages, but ready simulation is better behaved in some process algebraic settings, such as CCS [27].

These various notions of simulation can be compared: in particular, complete strong simulation (with parameters $(\mathcal{D}, \mathcal{F}, \mathcal{L})$ = (`nodelay`, `nofreeze`, `lock`)) is the strongest, and weak simulation (with parameters $(\mathcal{D}, \mathcal{F}, \mathcal{L})$ = (`delay`, `freeze`, `nolock`)) is the weakest.

Lemma 4.1 (Comparing the various notions of simulation [`Sims.v:796`]). `nofreeze` $\Longrightarrow$ `freeze_div` $\Longrightarrow$ `freeze`. `nodelay` $\Longrightarrow$ `delay`. `lock` $\Longrightarrow$ `nolock`.

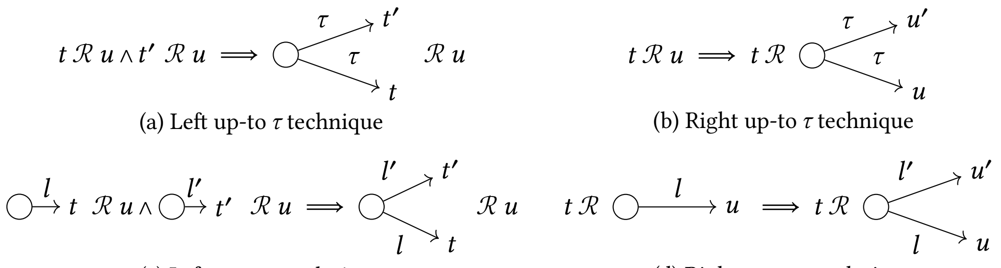
4.2 Asymmetric up-to Techniques

This section defines several up-to techniques for parameterized μdiv-simulation. As a reminder, up-to techniques are theorems that transform a simulation up-to candidate, which is the setting we are working in. The particular case of transitivity is addressed in Section 4.3. The Choice Trees case study (Section 5) will define a few more specialized up-to techniques.

The present section focuses on asymmetric up-to techniques, that act either on the left-hand side or on the right-hand side of $\mathcal{R}$. Figure 11 sums up the four up-to techniques that this section defines, on small examples. These techniques are meant for backward reasoning: they transform a proof goal into one or several smaller proof goals.

4.2.1 Left and Right up-to tau. These up-to techniques have already been discussed in Section 3.3. They can be generalized to parameterized simulation, with additional conditions.

For the divergence-sensitive variants of simulation (i.e., $\mathcal{F}$ = `freeze_div`), the left up-to tau technique allows to consume a τ transition from the left state without any answer from the right state of the simulation and without a divergence preservation side condition, which is precisely what is allowed in the simulation game when $\mathcal{F}$ = `freeze`. This provides a nice framework for formulating the left up-to tau technique, as a more relaxed variation of the up-to simulation step technique.


$t\,\mathcal{R}\,u \wedge t'\,\mathcal{R}\,u \implies$
τ
t'
τ
t
$\mathcal{R}\,u$
$t\,\mathcal{R}\,u \implies t\,\mathcal{R}$
τ
u'
τ
u
l
$t\ \mathcal{R}\,u \wedge$
l'
$t'\ \mathcal{R}\,u \implies$
l'
t'
l
t
$\mathcal{R}\,u$
$t\,\mathcal{R}$
l
$u \implies t\,\mathcal{R}$
l'
u'
l
u

(a) Left up-to τ technique (b) Right up-to τ technique

(c) Left up-to ϵ technique (d) Right up-to ϵ technique

Fig. 11. Examples for the four asymmetric up-to techniques. States t, t', u, u' can have arbitrary outgoing transitions.

Lemma 4.2 (Relaxed up-to simulation step for µdiv-simulation[Sims.v:915]). *If $\mathcal{R}$ is a $(\mathcal{D}, \texttt{freeze\_div}, \mathcal{L})$-simulation candidate, then for all states t and u,* $\texttt{simF}_{(\mathcal{D},\texttt{freeze},\mathcal{L})}\ t\,u \implies t\,\mathcal{R}\,u$.

As a consequence of this result, in a simulation proof with $\mathcal{F} = \texttt{freeze\_div}$, only one divergence-preserving simulation step is needed, then regular simulation steps are sufficient. Intuitively, the reason is that these additional simulation steps will at worst consume a finite number of τ transitions on the left-hand side before tying the coinductive knot. For this to be safe, the right-hand side needs to also perform at least one transition, which is ensured by the divergence-preserving step.

The right up-to tau technique consumes τ transitions on the right-hand side. Because it consumes more τ transitions on the right-hand side than on the left-hand side, it naturally requires $\mathcal{D} = \texttt{delay}$.

Theorem 4.3 (Right up-to tau technique [Sims.v:453]). *If $\mathcal{R}$ is a $(\texttt{delay}, \mathcal{F}, \mathcal{L})$-simulation candidate, then $\forall t\,u\,u', u \xrightarrow{\tau^*} u' \wedge t\,\mathcal{R}\,u' \implies t\,\mathcal{R}\,u$..*

4.2.2 *Left and Right up-to epsilon.*

Up-to epsilon techniques "reduce choice" by removing some transitions from an LTS. These are a generalization of the "up-to epsilon" techniques of Choice Trees [12], but also of the built-in asynchronous progress techniques of FreeSim [15].

The left up-to epsilon technique splits t into a partition S of "sub-states" with less outgoing transitions, in order to split the simulation proof into smaller pieces.

Theorem 4.4 (Left up-to epsilon technique [Sims.v:562]). *Let t and u be states, S a set of states, and $\mathcal{R}$ a simulation candidate with $\mathcal{L} = \texttt{nolock}$.*[9]

$$\left(\left(\forall l t', t \xrightarrow{l} t' \implies \exists s \in S, s \xrightarrow{l} t'\right) \wedge \forall s \in S, s\,\mathcal{R}\,u\right) \implies t\,\mathcal{R}\,u$$

The right up-to epsilon technique removes some of the outgoing transitions from u.

Theorem 4.5 (Right up-to epsilon technique [Sims.v:575]). *Let $\mathcal{R}$ be a simulation candidate with $\mathcal{L} = \texttt{nolock}$.*

$$\forall t\,u\,u', \left(\left(\forall l u'', u' \xrightarrow{l} u'' \implies u \xrightarrow{l} u''\right) \wedge t\,\mathcal{R}\,u'\right) \implies t\,\mathcal{R}\,u$$

[9]Both up-to epsilon theorems can be adapted to deadlock-sensitive simulation, but this requires additional hypotheses. We omit the details, see our Rocq development for the exact statement.

These two techniques are more general than their up-to τ counterparts and could subsume them, but they operate with the additional assumption that removing some of the outgoing transitions transitions from a state gives another valid state of the LTS. This is to some extent the case for Choice Trees, but not for CompCert. For the latter, the up-to epsilon techniques are not directly applicable.

4.3 Transitivity and Rewriting Similar Terms

In a verified compiler, it is crucial to have a notion of simulation that is *transitive*, as it allows composing proofs of simulation for the various compilation passes into the end-to-end proof that the source program simulates the final compiled program. Our parameterized simulation is indeed transitive for all the possible values of parameters.

THEOREM 4.6 (TOP-LEVEL TRANSITIVITY [`Sims.v:752`]). $\mathtt{sim}_{(\mathcal{D},\mathcal{F},\mathcal{L})}$ *is transitive for all the possible values of* $\mathcal{D}$, $\mathcal{F}$, *and* $\mathcal{L}$.

In a coinduction up-to setting, one can hope to go further than this "top-level" result about `sim`, and prove a transitivity result about simulation candidates $\mathcal{R}$. Because results about coinduction candidates can be transported for free to their greatest fixpoints, such an up-to technique implies the above top-level transitivity result, and it has additional benefits. A major use case of up-to transitivity is that it allows rewriting similar terms during simulation proofs: if one has already established that $\mathtt{sim}\ a\ b$ and now desires to prove that $a\ \mathcal{R}\ c$, it suffices to prove that $b\ \mathcal{R}\ c$. This special case is called *up-to similarity*.

The validity of up-to transitivity depends on the values of the parameters.

THEOREM 4.7 (UP-TO TRANSITIVITY [`Sims.v:1160`]). *Let* $\mathcal{R}$ *be a* $(\mathtt{nodelay}, \mathcal{F}, \mathcal{L})$*-simulation candidate* $\mathcal{R}$. *If furthermore* $\mathcal{F} \neq \mathtt{freeze\_div}$, *then* $\mathcal{R}$ *is transitive.*

As we will now justify, the up-to transitivity technique is unsound for the notions of simulation not covered by the above theorem, in particular the weak variants of similarity. This is not surprising, as this limitation had been noted for weak bisimulation several decades ago [37].

Why up-to transitivity does not hold. Let us try to prove the following weak simulation result, with t and u arbitrary states: $\mathtt{sim}_{(\mathtt{delay},\mathtt{freeze},\mathtt{nolock})}\ t\ u$. Our initial proof goal is $t\ \mathcal{R}\ u \vdash \mathtt{simF}\ \mathcal{R}\ t\ u$. Assuming up-to transitivity holds, it reduces to $t\ \mathcal{R}\ u \vdash \mathtt{simF}\ \mathcal{R}\ t \left(\bigcirc \xrightarrow{\tau} t\right) \wedge \mathtt{simF}\ \mathcal{R} \left(\bigcirc \xrightarrow{\tau} t\right) u$. The left proof obligation is trivially verified because the LTSs are weak-similar. As for the right one, the left τ transition can be consumed, leading to $t\ \mathcal{R}\ u \vdash t\ \mathcal{R}\ u$, which concludes the proof. Weak similarity up-to transitivity would thus relate all the states with each other: it does not hold.

Fortunately, several strategies have been devised in the setting of weak bisimulation up-to [32, 34, 37] to still establish restricted statements of up-to transitivity. In the following, we adapt two such strategies to our parameterized simulation: the expansion preorder, and the distinction between τ and non-τ transitions.

Theorem 4.8 is a direct adaptation of the expansion preorder used for weak bisimulation. As we have seen, the main challenge with weak (bi)simulation up-to is that in the general case it could introduce τ transitions in the left state that could then be exploited to consume the coinductive guard and trivialize any (bi)simulation proof. A possible answer is to rewrite with a variant of simulation that does not permit to introduce more τ transitions on the left. In the case of weak *bi*simulation, the expansion preorder can fill that role. In our case, parameterized µdiv-simulation has a flag that exactly does that: the *delay* flag, which suggests a restricted statement of up-to similarity for the left-hand side.

THEOREM 4.8 (LEFT UP-TO SIMILARITY [Sims.v:1120]). *If $\mathcal{R}$ is a $(\mathcal{D}, \mathcal{F}, \mathcal{L})$-simulation candidate, then:* $\forall t t' u, \mathtt{sim}_{(\mathtt{nodelay}, \mathcal{F}, \mathcal{L})}\ t\ t' \wedge t'\ \mathcal{R}\ u \implies t\ \mathcal{R}\ u$.

For instance, during a proof of weak simulation up-to companion, the above theorem allows rewriting terms that at least *option-similar* (as per Table 1) on the left-hand side. It is also possible to rewrite terms that are *strongly similar*, as it is even stronger than option similarity.

There is an analogous additional difficulty for *divergence-sensitive* notions of simulation: rewriting on the right-hand side could add τ transitions that could then be artificially used to trivially prove divergence preservation. Again, a possible answer is to rewrite with a variant of simulation that does not permit to introduce more τ transitions on the right. Our parameterized simulation has a flag that does exactly that: the *freeze* flag, which suggests a restricted statement of up-to similarity for the purpose of rewriting terms on the right-hand side.

THEOREM 4.9 (RIGHT UP-TO SIMILARITY [Sims.v:1013]). *If $\mathcal{R}$ is a $(\mathcal{D}, \mathcal{F}, \mathcal{L})$-simulation candidate with $\mathcal{F} \neq \mathtt{freeze\_div}$, then:*

$$\forall t u' u, t\ \mathcal{R}\ u' \wedge \mathtt{sim}_{(\mathcal{D}, \mathcal{F}, \mathcal{L})}\ u'\ u \implies t\ \mathcal{R}\ u$$

Alternatively, if $\mathcal{R}$ is a $(\mathcal{D}, \mathtt{freeze\_div}, \mathcal{L})$-simulation candidate, then:

$$\forall t u' u, t\ \mathcal{R}\ u' \wedge \mathtt{sim}_{(\mathcal{D}, \mathtt{nofreeze}, \mathcal{L})}\ u'\ u \implies t\ \mathcal{R}\ u$$

Both on the left and right sides, the problems with rewriting similar terms comes from τ transitions. This suggests that, in states with no outgoing τ transition, unrestricted rewriting is possible. Following [34], we can tweak the simulation game in Figure 10 so that the visible l diagram leads to $\mathcal{R}^+$ instead of $\mathcal{R}$, essentially hardcoding transitivity into the simulation diagram. With this definition, we get one more up-to technique for rewriting similar terms.

THEOREM 4.10 (UP-TO SIMILARITY FOR NON-TAU STATES [Sims.v:991]). *If $\mathcal{R}$ is a $(\mathcal{D}, \mathcal{F}, \mathcal{L})$-simulation candidate, then for all states t t' u u' such that t and t'* have no outgoing τ transitions,

$$\mathtt{sim}_{(\mathcal{D}, \mathcal{F}, \mathcal{L})}\ t\ t' \wedge t'\ \mathcal{R}\ u' \wedge \mathtt{sim}_{(\mathcal{D}, \mathcal{F}, \mathcal{L})}\ u'\ u \implies t\ \mathcal{R}\ u$$

This time, there is no restriction on the parameters of the simulation game, and rewriting is allowed both on the left and right sides.

Throughout this section, we have defined various up-to techniques for our parameterized simulation, to allow for more compositional reasoning. In particular, Theorems 4.2, 4.8, and 4.9 were heterogeneous in the values of the simulation parameters, which justifies our use of a parameterized definition.

5 Application: Choice Trees

Until now, we made few assumptions about the LTSs we are working with. This relatively generic setting was still sufficient to define various notions of simulation and prove a number of properties about them. This section finally instantiates our theory to a more concrete setting of monadic interpreters. Interaction Trees (ITrees) [47] are a data structure for representing and reasoning about recursive and impure programs that has gained a lot of attention these past few years, including for verified compilation projects [48]. ITrees can be seen as structured deterministic LTSs shallowly embedded in Rocq. Various extensions to nondeterministic LTSs have then been proposed [3, 10, 13, 15, 39]. This section focuses on Choice Trees 2.0 (CTrees) [12] because among these notions they capture the widest class of LTSs: to our knowledge they are the only ITree-like structure that can capture the LTSs in Figure 1. This generality has its drawbacks [3] but provides us with an interesting setting.

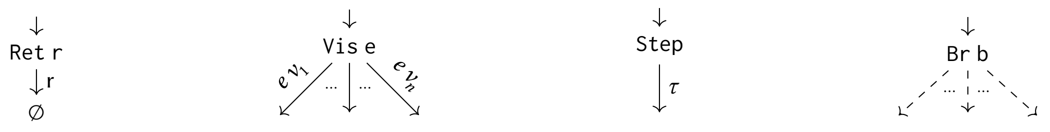


Fig. 12. The interpretation of the four kinds of nodes of values of type `ctree E B X` in terms of a labelled transition system. Dashed transitions represent an inductive search down the tree for a transition.

One of the intended use cases of CTrees is the formalization of the semantics of programming languages. In particular, they have been used to model a small concurrent subset of the semantics of LLVM IR [14]. The library notably includes several notions of simulation and bisimulation based on standard process algebraic definitions, with several coinduction up-to techniques. But as of now, the CTrees library focuses more on notions of *strong* (bi)simulation, and does not come with a formalization of divergence-sensitive weak simulation. In this section, we aim to instantiate our parameterized simulation on CTrees and derive a proof system for our parameterized simulation on CTrees, in order to augment them with a notion of divergence-sensitive weak simulation.

5.1 Basic Definitions

Choice Trees are a coinductive model for the semantics of programs, in particular concurrent programs, defined in Rocq as follows:

```
CoInductive ctree (E B : Type -> Type) (R : Type) :=
| Ret (r : R) (* pure computation *)
| Vis {X : Type} (e : E X) (k : X -> ctree) (* external event *)
| Br {X : Type} (c : B X) (k : X -> ctree) (* delayed branching *)
| Step (t : ctree) (* internal computation *)
```

As of CTrees 2.0 [12], an object of type `ctree E B X` is a possibly infinite tree that can contain four kinds of nodes that each have their own semantics, as defined above.[10]

- `Ret` nodes represent the final return of a value of type X at the end of an execution. They have no successors.
- `Vis` nodes represent effects: the program emits an event $e \in E\ X$ to the environment in accordance with the signature of events E, and the environment answers with a value of type X.
- `Step` nodes represent internal computations, they play the same role as τ transitions.
- `Br` nodes represent nondeterministic choices, emitted in accordance with the signature B. Two special cases of `Br` have specific notations for convenience: `Stuck` is a nullary choice, `Guard` is a unary choice.

Figure 12 depicts a straightforward injection of CTrees into LTSs: `Ret` nodes generate a single transition into a stuck state, labelled with the return value. `Vis` nodes generate one transition per possible answer from the environment, each labelled with both the request and the answer. `Step` nodes generate a τ transition. `Br` nodes do not generate any transitions, they are collapsed in the LTS.

From this injection, the CTrees library defines in particular strong simulation and complete simulation on the LTSs underlying CTrees, and defines up-to techniques and a proof system for both of these notions of simulation on CTrees. However, we are not aware of any effort to develop

[10]The first iteration of CTrees [13] was limited to `Br` nodes of finite arity, and contained n-ary `Step` nodes called `BrS` that can now be encoded as `Br` followed by `Step`.

$$\frac{}{\mathtt{Ret}\ v\ \mathcal{R}\ \mathtt{Ret}\ v}\ \texttt{(ret)} \qquad \frac{\forall x \in X, (k\ x)\ \mathcal{R}\ u \quad \mathcal{L} = \mathtt{nolock} \lor X\ \text{inhabited}}{(\mathtt{Br}\ b\ k)\ \mathcal{R}\ u}\ \texttt{(br\_l)}$$

$$\frac{\exists y, t\ \mathcal{R}\ (k'\ y) \quad \mathcal{L} = \mathtt{nolock} \lor k'\ y \to \lor (\mathcal{F} = \mathtt{nofreeze} \land t \to)}{t\ \mathcal{R}\ (\mathtt{Br}\ b\ k')}\ \texttt{(br\_r)}$$

$$\frac{t\ \mathcal{R}\ u \quad \mathcal{D} = \mathtt{delay}}{t\ \mathcal{R}\ \mathtt{Step}\ u}\ \texttt{(step\_r)} \qquad \frac{t\ \mathcal{R}\ u \quad \mathcal{F} = \mathtt{freeze} \lor (\mathcal{F} = \mathtt{freeze\_div} \land \mathtt{divpres}\ t\ u)}{\mathtt{simF}\ \mathcal{R}\ (\mathtt{Step}\ t)\ u}\ \texttt{(step\_l)}$$

$$\frac{t\ \mathcal{R}\ u \quad \mathcal{F} = \mathtt{freeze\_div}}{\mathtt{Step}\ t\ \mathcal{R}\ u}\ \texttt{(step\_l')} \qquad \frac{t\ \mathcal{R}\ u}{\mathtt{simF}\ \mathcal{R}\ (\mathtt{Step}\ t)\ (\mathtt{Step}\ u)}\ \texttt{(step)}$$

$$\frac{\forall v,\ (k\ v)\ \mathcal{R}\ (k'\ v)}{\mathtt{simF}\ \mathcal{R}\ (\mathtt{Vis}\ e\ k)\ (\mathtt{Vis}\ e\ k')}\ \texttt{(vis)} \qquad \frac{t \not\to \land (\mathcal{L} = \mathtt{nolock} \lor u \not\to)}{t\ \mathcal{R}\ u}\ \texttt{(stuck)}$$

Fig. 13. Proof rules for coinductive proofs of parameterized simulation on CTrees[`CTree.v:451`]

a notion of divergence-sensitive weak simulation for CTrees,[11] despite its crucial role for verified compilation. By instantiating our parameterized simulation on CTrees, we can vastly extend the set of supported notions of simulation on CTrees, and develop a common parameterized equational theory for all of these simulations. To instantiate parameterized simulation on CTrees, we need to provide a method to build an LTS from a CTree. Because the CTree library already defines such an injection, this is immediate [`CTree.v:31`].

5.2 A Proof System

Due to the structure inherent to CTrees, it is possible to reason equationally about simulations that involve CTrees. The CTrees library defines proof systems for strong simulation and for complete simulation, with some duplication between them. This section defines a proof system with rules conditioned on the values of the simulation parameters, yielding a concise unified theory for parameterized simulation. These rules are depicted in Figure 13. In this figure, $t \to$ means that there exists at least one (possibly τ) transition from t.

This proof system is meant to be used during coinductive proofs of simulation, the rules act on a parameterized simulation candidate $\mathcal{R}$. Just like up-to techniques, these results can trivially be transported to top-level similarity, by replacing both $\mathcal{R}$ and `simF` $\mathcal{R}$ by `sim` in the rules.

The `(ret)`, `(step)`, and `(vis)` rules are straightforward: they match the constructors of both sides and consume the coinductive guard. As for the other rules, they are direct consequences of up-to techniques and simulation diagrams of parameterized simulation. The `(step_l)` rule is a consequence of the `freeze` and `freeze_div` diagrams from Figure 10. The `(step_l')` rule is a consequence of Lemma 4.2. The `(step_r)` rule is a consequence of Theorem 4.3. The `(br_l)` rule is a consequence of Theorem 4.4. The `(br_r)` rule is a consequence of Theorem 4.5. The `(stuck)` rule is a consequence of the `lock` diagram from Figure 10.

These rules, proved in Rocq, provide sound proof systems for coinductive proofs of the various variants of simulation covered by our parameterized definition. It is however not complete, but one can always write a simulation proof that mixes the use of these rules and lower-level results when needed.

[11]The "strong simulation on the Br-aware LTS" is arguably a notion of weak simulation, but it is not divergence-sensitive.

5.3 Up-to Bind

The above rules act on CTree constructors, but CTrees can also be built using various combinators. Several have been defined and studied. This section focuses on `bind` and `iter`, and establishes expected properties to reason about them in parameterized simulation proofs. These two combinators are particularly interesting because they respectively encode sequencing and loops,

The monadic `bind : ctree E B X -> (X -> ctree E B Y) -> ctree E B Y` operator takes a CTree t, a continuation k, and replaces every node of the form (`Ret` x) in the CTree with the continuation $k\ x$. It is especially useful to represent sequencing computations. As with ITrees and similar structures, a proof of simulation can be split into two at a `bind` through the use of an up-to technique.

Theorem 5.1 (Up-to bind [CTree.v:328]). *Let $\mathcal{R}$ be a $(\mathcal{D}, \mathcal{F}, \mathcal{L})$-simulation candidate with $\mathcal{D} = \texttt{delay} \vee \mathcal{F} \neq \texttt{freeze\_div}$. If t, t' are CTrees and k, k' are continuations, then:*

$$((\mathcal{L} = \texttt{nolock} \vee (\forall x, k\ x \rightarrow \wedge\ k'\ x \rightarrow)) \wedge \texttt{sim}\ t\ t' \wedge (\forall x, k\ x\ \mathcal{R}\ k'\ x)) \implies \texttt{bind}\ t\ k\ \mathcal{R}\ \texttt{bind}\ t'\ k'$$

The restriction on $\mathcal{D}$ and $\mathcal{F}$ is not because of a corner case that makes the result not hold for divergence-sensitive option simulation. We actually believe that it does also hold in that case, but that we are lacking the right tools to establish this result for the moment. Our current proof for the up-to bind technique is already quite complicated (a few hundred lines counting auxiliary lemmas when our other proofs are all below 100 lines) so we leave this to future work. Still, the proof as it is now establishes an up-to bind technique for 10 notions of simulation. Our parameterized definition was key to avoid major code duplication between different proofs of up-to bind techniques.

A notable corollary of the up-to bind technique is that similarity is preserved by the iteration combinator. There are two distinct iteration combinators defined on CTrees: one that guards recursion with a unary `Br` node, and one that guards recursion with a `Step` node. We focus on the second one, which is more standard and matches the iteration combinator from ITrees. Its signature is `iterS {I: Type} (body : I -> ctree E B (I + X)) (i : I) : ctree E B X`.

The given body of the loop is initially executed with the given i parameter. The body can return a value in `I`, in which case the loop is re-entered, or in `X`, in which case the loop is exited and the value is returned. Because `iterS` is defined in terms of `bind`, its compatibility with similarity is a direct consequence of Theorem 5.1. It inherits the same requirements on the values of the parameters $\mathcal{D}$ and $\mathcal{F}$.

Corollary 5.2 (`iterS` preserves similarity [CTree.v:606]). *If $\mathcal{D} = \texttt{delay} \vee \mathcal{F} \neq \texttt{freeze\_div}$, then:* $\forall i, \texttt{sim}_{(\mathcal{D},\mathcal{F},\mathcal{L})}\ (\texttt{body}\ i)\ (\texttt{body'}\ i) \implies \forall i, \texttt{sim}_{(\mathcal{D},\mathcal{F},\mathcal{L})}\ (\texttt{iterS body}\ i)\ (\texttt{iterS body'}\ i)$.

This result is valuable in that it allows to reason about possibly diverging programs *at the top-level* using only `sim` and not a simulation candidate, as long as the possible divergence comes from the `iterS` combinator. This avoids the need for a coinductive proof in sufficiently simple settings, typically sequential programs.

6 Application: A CompCert Pass

CompCert [8, 22] is a major verified compilation project. It contains many successive verified compilation and optimization passes from CompCert C, a language close to ISO C 2011 [20], down to assembly code. All the intermediate representations are formalized as LTSs (through small-step operational semantics), and all the CompCert passes come with a behavioral inclusion result, more specifically an explicit normed simulation result.[12] The very first intermediate representation of

[12] Some of these explicit normed simulation results are forward simulations, see our paragraph further down in this section.

CompCert is nondeterministic, then most of the LTSs considered are deterministic, which makes for simpler proofs of simulation. This section instantiates μdiv-simulation on CompCert. We first give our motivation for this application, before discussing the instantiation and an actual proof of an existing compilation pass.

Motivation: `match_states`*.* Of course, the explicit normed simulation of CompCert is not based on modern coinduction up-to libraries, as they did not exist at the beginning of the CompCert project. Rather, it is defined propositionally, in the style of Definition 2.1. As a consequence, simulation relations (typically named `match_states` in CompCert) have to be entirely characterized from the start in proofs of simulation. Some of these `match_states` are quite complex, spanning several dozen lines of code. They are sometimes even longer than the relational description of the transformation being proved.[13] Because coinduction up-to allows to only partially specify the simulation relation, its use could partially alleviate this problem. But coinduction up-to is not a silver bullet, there is some complexity inherent to large-scale semantics such as that of CompCert.

Motivation: The explicit norm. Most proofs of normed simulation in CompCert contains not only a simulation relation, but also a measure on states, because CompCert relies on explicit normed simulation. This explicit measure can be uncomfortable to work with, and a modern alternative could alleviate this burden. Furthermore, normed simulation is not complete with respect to divergence-sensitive weak simulation, unlike our μdiv-simulation. This is however less relevant for CompCert as long as its intermediate representations are fully deterministic, because normed simulation and divergence-sensitive weak simulation do coincide in a deterministic setting.

Motivation: Up-to techniques and eventually simulation. Eventually simulation is a proof technique for normed simulation introduced a few years ago in CompCert.[14] In general terms, it transforms a `match_states` relation so that a finite number of τ transitions can be skipped on the left-hand side of the simulation, which makes it equivalent to our left up-to τ technique. This eventually simulation is in fact an up-to technique, because an up-to technique is precisely a function that transforms a simulation candidate. However, it is currently defined propositionally, outside of any coinduction up-to framework, which makes things harder: up-to techniques have to be explicitly integrated in the simulation relations of proofs that use them, and they cannot be combined at will. Indeed, simulation diagrams in CompCert are roughly of the shape "after one simulation step we get back to the `match_states` relation", but diagrams of proofs that make use of eventually simulation are of the shape "after one simulation step we get back to the relation obtained by applying the `eventually` transformer to the `match_states` relation". Using the companion here, as we did throughout this paper, would both lighten proofs that make use of such up-to techniques, and make them usable at no additional cost in other simulation proofs.

The instantiation. Because CompCert is already based on LTSs, instantiating our theory on it is, once again, effortless, taking only about 20 lines of Rocq [`CompCert.v:30`].

Common subexpression elimination. We show the applicability of our simulation on CompCert by porting an existing simulation proof to use it, specifically the proof of the common subexpression elimination (CSE) pass, an optimization pass that factors out redundant expressions. In its current state in CompCert, it uses the eventually simulation technique. With μdiv-simulation, this is not necessary because the equivalent left up-to τ technique has already been proved in a generic setting and can be invoked on the fly during the proof. Overall, we modified a few dozen lines in this

[13]See for instance the relation for the inlining pass: https://github.com/AbsInt/CompCert/blob/0eabf20d971e4fd0c7c92dad8a171af0c0fe3ed7/backend/Inliningproof.v#L858

[14]https://github.com/AbsInt/CompCert/blob/0eabf20d971e4fd0c7c92dad8a171af0c0fe3ed7/common/Smallstep.v#L773

300-line proof to port it to µdiv-simulation[🐓 `CompCert.v:166`]. The proof structure is largely the same as before, but it now occasionally invokes the `upto_E0_l` lemma (a specialization of the left up-to τ technique to CompCert) instead of exploiting a hardcoded eventually simulation technique.

Forward simulations. Forward simulation [25] is a proof technique that consists of establishing that the compiled program simulates the source program. This is the "wrong" direction for verified compilation, but under some conditions (namely, source receptiveness and target determinacy) forward simulation implies backward simulation. Because these conditions are verified in most of the compilation chain of CompCert, this proof technique is extensively used, including in the proof of the CSE pass. We proved that the forward simulation proof technique is valid for µdiv-simulation on arbitrary LTSs, under the same assumptions as in CompCert [🐓 `Determinism.v:200`]. In CompCert, forward and backward simulations are defined separately, because of subtleties in the handling of UBs. In our case, we use µdiv-simulation in both directions, but *on different LTSs*: UB states have to be dualized into empty states in the forward direction. Our proof proceeds similarly to the proof in CompCert, but it does not explicitly build a backward measure from a forward measure (thanks to the implicit characterization of divergence preservation), nor a backward relation from a forward relation (thanks to coinduction up-to),

Discussion. We eluded many internal details of CompCert for brevity, such as the specific handling of undefined behaviors (it poses no particular challenge). Our goal in this section was to demonstrate that our formalism can be instantiated on a realistic verified compiler, and that doing so does not introduce additional complexity. On the contrary, it can potentially simplify simulation proofs. We did not push this case study further, porting more compilation passes or other parts of the CompCert codebase would be a tedious endeavour with little benefit. Rather, we believe µdiv-simulation could be a valuable additional tool for the development of future compilation passes, even more so if someday more CompCert intermediate representations are made nondeterministic.

7 Related Work

Freely-stuttering simulation. Freely-stuttering simulation, or FreeSim [15], is a notion of simulation that attempts to combine the strengths of implicit normed simulation and explicit normed simulation. The Rocq formalization of FreeSim relies on a relatively abstract setting of structured LTSs in order to make it usable in different verified compilation projects, with two case studies: DTrees (a data structure similar to CTrees) and CompCert. On the flip side, these various aims of FreeSim ultimately make the definition of freely-stuttering simulation quite complicated: it relies on a combination of induction, coinduction and two decreasing measures. For a more precise definition we refer the interested reader to the introductory FreeSim paper. Our definition of µdiv-simulation is arguably simpler, and is furthermore complete with respect to divergence-sensitive weak simulation.

Another limitation of FreeSim is that it can only be applied on LTSs of a very specific shape: LTSs in which any state that has an outgoing observable transition has no other outgoing transition at all. This is not the case for the LTSs in CompCert, so the FreeSim development uses another definition of FreeSim for their CompCert case study, further complexifying the development. The FreeSim paper claims that any LTS can be transformed to an LTS of the expected shape, but does not formally prove it. We propose such an LTS transformer in our development [🐓 `FreeSim.v:20`] in order to prove that FreeSim on a "FreeSimized" LTS[15] implies our implicit normed simulation

[15]We actually do not consider FreeSim itself, but FreeSim's variant of implicit normed simulation, proved equivalent to FreeSim in its introductory paper.

on a standard LTS [`FreeSim.v:343`]. Because implicit normed simulation is itself strictly more restrictive than µdiv-simulation, our notion of simulation is thus more complete than FreeSim.

FreeSim has two strengths in comparison with µdiv-simulation: it supports angelic nondeterminism, which is out of the scope of the current paper (it does not fit into standard LTSs), and it has a mechanism for remembering asynchronous progress. This second strength can be emulated by our up-to epsilon techniques in simple cases, and it could be reproduced in the general case using up-to techniques without changing our definition of µdiv-simulation, but we leave this to future work.

Weak-tau simulation. CompCertTSO was a successful (but now unmaintained) project to extend CompCert to concurrent programs [45]. As is the case in CompCert, its compilation passes are proved correct using explicit normed simulation results, *with one exception.* On top of extending CompCert's compilation passes to concurrent programs, CompCertTSO adds three passes related to redundant fence elimination, an optimization specific to concurrent programs. In one of these three passes, the CompCertTSO authors tried to prove a normed simulation result for two months but failed, presumably because it is not complete enough. In the end, the CompCertTSO authors designed a novel notion of simulation specifically for this proof: *weak-tau simulation* [41]. A weak-tau simulation is a pair of mutually-defined relations of simulation and divergence preservation, making this notion quite close to µdiv-simulation in spirit (though formulated in a different style). Yet we are confident that it is not complete with respect to divergence-sensitive weak simulation,[16] because the divergence preservation relation of weak-tau simulation follows the same basic idea as ours but *without the inductive case on the right of Figure 7.* This slight omission has major consequences: a simulation result as simple as `sim` $\left(\bigcirc \xrightarrow{\tau} t\right)$ t becomes unprovable. For the same reason, the states t_3 and u_3 of our introductory example from Figure 1 are not weak-tau similar, which implies that t_0 and u_0 are not either.

Divergence sensitivity. Divergence-sensitive weak bisimulation is studied in [24], and a verification method is devised for it, dubbed *inductive weak bisimulation.* While it could certainly be adapted to divergence-sensitive weak simulation, this verification method relies on first establishing a weak bisimulation and only then verifying divergence preservation, a proof strategy that is not viable in a coinduction up-to setting.

Parameterized simulation. We are not aware of any pre-existing notion of parameterized simulation, however notions of parameterized bisimulation have been proposed. In 1993, van Glabbeek famously compared 155 notions of bisimulation [44] thanks to carefully parameterized definitions. More recently, [49] proposed a Rocq formalization of a parameterized notion of bisimulation that covers weak and strong bisimulation as well as the expansion preorder [1]. This Rocq formalization contains a study of up-to techniques for these three notions of bisimulation. It is however restricted to deterministic LTSs.

Simulations in verified compilation. Actual verified compilation projects are usually based on deterministic LTSs [21, 22, 48] because it makes for simpler proofs. As is visible in our discussion about CompCertTSO, things get more complicated in a concurrent setting. Beyond divergence sensitivity, other dimensions are worthy of interest. For instance, [23] defines a notion of refinement for concurrent programs that separates traces depending on memory access locations, yielding a notion weaker than usual notions of simulation, but this is orthogonal to our contribution.

[16] Note however that we have not formalized any result about weak-tau simulations in our Rocq development.

8 Discussion

This paper introduced μdiv-simulation as a powerful notion of simulation equivalent to classical divergence-sensitive weak simulation, and proposed various up-to techniques. Because we maintained a relatively abstract setting in Sections 3 and 4, our study can be applied to various verified compilation projects, including CompCert, and could also be applied to process algebras and model checking all the same. Our claims are formalized in Rocq, and we strived to keep the core theory around our parameterized simulation[`Sims.v:428`] clean to make it more approachable. Nearly all of our proofs about generic parameterized simulation are below 30 lines of Rocq, confirming the power of the coinduction up-to companion proof approach. Beyond our technical contributions, one goal of this paper is to increase awareness about coinduction up-to in the verified compilation community, in particular through Section 6. We based our work on modern coinduction up-to companion and tower induction, but the later notion of *diacritical companion* [6] could possibly simplify some of our proofs around up-to techniques.

Design note: LTS vs simulation game. Some features can be equivalently encoded either in LTSs or in the simulation game, such as deadlock sensitivity (encoded as a parameter in our simulation game) or undefined behaviors (that we prefer to leave to the LTS). Both have benefits and drawbacks, time will tell if our design choices in this regard were the right ones.

Perspectives: beyond simulations. A possible ambitious extension of our work would be to mechanize and study a parameterized notion of bisimulation. Because there are many ways to symmetrize a simulation functor into a bisimulation functor, this could quickly become unmanageable [44]. A possible starting point could be to combine the recently-proposed notion of *intertwined bisimulation* [12] (basically a delay-style variant of $\approx_\tau$-bisimulation [36]) with μdiv-simulation into a divergence-sensitive notion of bisimulation for verified compilation. Because bisimulation is significantly less relevant than simulation in verified compilation, we did not explore this perspective. Another valuable extension could be to go beyond simulations and develop a coinductive characterization of trace inclusion that handles divergence preservation in the spirit of μdiv-simulation. Again, we leave this to future work.

Data-Availability Statement

The Rocq development for this paper is available on Zenodo [11] and released on opam as rocq-sims.

Acknowledgements

I am grateful to Sylvain Boulmé and David Monniaux for various discussions, valuable feedback, and insights about CompCert. I thank Benjamin Bonneau for interesting discussions about simulations, and for the programs of Figure 5. I thank Yannick Zakowski and Ludovic Henrio for valuable feedback. Finally, I thank the POPL'26 reviewers for their reviews and suggestions that helped improve this paper.

References

[1] S. Arun-Kumar and M. Hennessy. 1992. An efficiency preorder for processes. *Acta Informatica* 29, 8 (01 Aug 1992), 737–760. https://doi.org/10.1007/BF01191894

[2] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. 1987. On the consistency of Koomen's Fair Abstraction Rule. *Theoretical Computer Science* 51, 1 (1987), 129–176. https://doi.org/10.1016/0304-3975(87)90052-1

[3] Patrick Bahr and Graham Hutton. 2023. Calculating Compilers for Concurrency. *Proc. ACM Program. Lang.* 7, ICFP, Article 213 (aug 2023), 28 pages. https://doi.org/10.1145/3607855

[4] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

[5] Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. A Two-Phase Infinite/Finite Low-Level Memory Model: Reconciling Integer–Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction. *Proc. ACM Program. Lang.* 8, ICFP, Article 263 (aug 2024), 29 pages. https://doi.org/10.1145/3674652

[6] Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. 2019. Diacritical Companions. *Electronic Notes in Theoretical Computer Science* 347 (2019), 25–43. https://doi.org/10.1016/j.entcs.2019.09.003 Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics.

[7] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-End. In *FM 2006: Formal Methods*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 460–475.

[8] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (01 Oct 2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

[9] B. Bloom, S. Istrail, and A. R. Meyer. 1988. Bisimulation Can't Be Traced. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/73560.73580

[10] Benjamin Bonneau. 2025. Relational reasoning on monadic semantics. In *36es Journées Francophones des Langages Applicatifs (JFLA 2025)*. Roiffé, France. https://doi.org/10.5281/zenodo.14508391

[11] Nicolas Chappe. 2025. *Rocq Artifact for "A Family of Sims with Diverging Interests"*. https://doi.org/10.5281/zenodo.17663346

[12] Nicolas Chappe, Paul He, Ludovic Henrio, Eleftherios Ioannidis, Yannick Zakowski, and Steve Zdancewic. 2025. Choice trees: Representing and reasoning about nondeterministic, recursive, and impure programs in Rocq. *Journal of Functional Programming* 35 (2025), e21. https://doi.org/10.1017/S0956796825100105

[13] 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

[14] Nicolas Chappe, Ludovic Henrio, and Yannick Zakowski. 2025. Monadic Interpreters for Concurrent Memory Models: Executable Semantics of a Concurrent Subset of LLVM IR. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Denver, CO, USA) *(CPP '25)*. Association for Computing Machinery, New York, NY, USA, 283–298. https://doi.org/10.1145/3703595.3705890

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

[16] 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. *Proc. ACM Program. Lang.* 6, POPL, Article 28 (Jan. 2022), 31 pages. https://doi.org/10.1145/3498689

[17] Hubert Garavel and Frédéric Lang. 2022. Equivalence Checking 40 Years After: A Review of Bisimulation Tools. In *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 13560)*, Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos (Eds.). Springer, 213–265. https://doi.org/10.1007/978-3-031-15629-8_13

[18] David Griffioen and Frits Vaandrager. 1998. Normed simulations. In *Computer Aided Verification*, Alan J. Hu and Moshe Y. Vardi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 332–344.

[19] 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)*. ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/2429069.2429093

[20] ISO/IEC. 2011. *ISO/IEC 9899:2011.*

[21] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019). https://doi.org/10.1017/S0956796818000229

[22] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (01 Dec 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[23] Liyi Li and Elsa L. Gunter. 2020. Per-Location Simulation. In *NASA Formal Methods*, Ritchie Lee, Susmit Jha, Anastasia Mavridou, and Dimitra Giannakopoulou (Eds.). Springer International Publishing, Cham, 267–287.

[24] Xinxin Liu, Tingting Yu, and Wenhui Zhang. 2017. Analyzing divergence in bisimulation semantics. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 735–747. https://doi.org/10.1145/3009837.3009870

[25] N. Lynch and F. Vaandrager. 1995. Forward and Backward Simulations. *Information and Computation* 121, 2 (1995), 214–233. https://doi.org/10.1006/inco.1995.1134

[26] R. Milner. 1981. A modal characterisation of observable machine-behaviour. In *CAAP '81*, Egidio Astesiano and Corrado Böhm (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 25–34.

[27] Robin Milner. 1989. *Communication and Concurrency.* Prentice-Hall, Inc., USA.

[28] Robin Milner and R.W. Weyhrauch. 1972. Proving compiler correctness in a mechanised logic. *Machine Intelligence* 7 (1972), 51–73.

[29] Keiko Nakata and Tarmo Uustalu. 2010. Resumptions, Weak Bisimilarity and Big-Step Semantics for While with Interactive I/O: An Exercise in Mixed Induction-Coinduction. In *Proceedings Seventh Workshop on Structural Operational Semantics, SOS 2010, Paris, France, 30 August 2010 (EPTCS, Vol. 32)*, Luca Aceto and Pawel Sobocinski (Eds.). 57–75. https://doi.org/10.4204/EPTCS.32.5

[30] Kedar S. Namjoshi. 1997. A simple characterization of stuttering bisimulation. In *Foundations of Software Technology and Theoretical Computer Science*, S. Ramesh and G. Sivakumar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 284–296.

[31] Lucian Popescu and Nuno P. Lopes. 2025. Exploiting Undefined Behavior in C/C++ Programs for Optimization: A Study on the Performance Impact. *Proc. ACM Program. Lang.* 9, PLDI, Article 161 (June 2025), 24 pages. https://doi.org/10.1145/3729260

[32] Damien Pous. 2007. New up-to techniques for weak bisimulation. *Theor. Comput. Sci.* 380, 1–2 (July 2007), 164–180. https://doi.org/10.1016/j.tcs.2007.02.060

[33] Damien Pous. 2016. Coinduction All the Way Up. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 307–316. https://doi.org/10.1145/2933575.2934564

[34] Damien Pous and Davide Sangiorgi. 2011. Enhancements of the bisimulation proof method. In *Advanced Topics in Bisimulation and Coinduction*, Davide Sangiorgi and Jan Rutten (Eds.). Cambridge University Press, 233–289.

[35] Damien Pous and Davide Sangiorgi. 2019. Bisimulation and Coinduction Enhancements: A Historical Perspective. *Formal Aspects of Computing* 31, 6 (01 Dec 2019), 733–749. https://doi.org/10.1007/s00165-019-00497-w

[36] Davide Sangiorgi. 2012. *Introduction to Bisimulation and Coinduction* (2nd ed.). Cambridge University Press, USA.

[37] Davide Sangiorgi and Robin Milner. 1992. The problem of "weak bisimulation up to". In *CONCUR '92*, W.R. Cleaveland (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 32–46.

[38] Steven Schäfer and Gert Smolka. 2017. Tower Induction and Up-to Techniques for CCS with Fixed Points. In *Relational and Algebraic Methods in Computer Science - 16th International Conference, RAMiCS 2017, Lyon, France, May 15-18, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10226)*, Peter Höfner, Damien Pous, and Georg Struth (Eds.). 274–289. https://doi.org/10.1007/978-3-319-57418-9_17

[39] Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott. 2023. Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations That Supports Auto-Active Verification. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:26. https://doi.org/10.4230/LIPIcs.ECOOP.2023.30

[40] The Rocq Development Team. 2025. *The Rocq Prover.* https://doi.org/10.5281/zenodo.15149629

[41] Viktor Vafeiadis and Francesco Zappa Nardelli. 2011. Verifying Fence Elimination Optimisations. In *Static Analysis*, Eran Yahav (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 146–162.

[42] Rob van Glabbeek. 2017. *A Branching Time Model of CSP.* Springer International Publishing, Cham, 272–293. https://doi.org/10.1007/978-3-319-51046-0_14

[43] Rob van Glabbeek, Bas Luttik, and Nikola Trčka. 2009. Branching Bisimilarity with Explicit Divergence. *Fundamenta Informaticae* 93, 4 (2009), 371–392. https://doi.org/10.3233/FI-2009-109 arXiv:https://doi.org/10.3233/FI-2009-109

[44] Rob J. van Glabbeek. 1993. The Linear Time - Branching Time Spectrum II. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR '93)*. Springer-Verlag, Berlin, Heidelberg, 66–81.

[45] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (jun 2013), 50 pages. https://doi.org/10.1145/2487241.2487248

[46] D.J. Walker. 1990. Bisimulation and divergence. *Information and Computation* 85, 2 (1990), 202–241. https://doi.org/10.1016/0890-5401(90)90048-M

[47] 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

[48] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (aug 2021), 30 pages. https://doi.org/10.1145/3473572

[49] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An Equational Theory for Weak Bisimulation via Generalized Parameterized Coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. https://doi.org/10.1145/3372885.3373813