Reversible Imperative Parallel Programs and Debugging

James Hoey(✉) and Irek Ulidowski

Department of Informatics, University of Leicester, Leicester, UK
{jbh11,iu3}@leicester.ac.uk

Abstract. We present a state-saving approach to reversible execution of imperative programs containing parallel composition. Given an original program, we produce an annotated version of the program that both performs forwards execution and all necessary state-saving of required reversal information. We further produce an inverted version of our program, capable of using this saved information to reverse the effects of each step of the forwards execution. We show that this process implements correct and garbage-free inversion. We give examples of how our implementation of reversible execution can be used for debugging, and demonstrate how a simulation tool we have developed for our approach can be used to examine the program state. Finally, we evaluate the performance and overheads associated with state-saving and inversion.

Keywords: Reversible computation · Debugging · Parallel composition · Imperative language · Inversion

1 Introduction

Reversible computation has been an area of increasing interest for many years. Reversible execution is the ability to undo the effects of running a program, and requires the majority of information to be preserved throughout the execution. This offers many benefits, including the suggestion within the Landauer principle [12] that not losing any information could lead to energy-efficient computation. Throughout this work we will explore the application to debugging.

An introduction into debugging and software bugs is provided by Zeller [28]. One common type of debugging, named *cyclic debugging*, is to run and re-run a program experiencing a bug. Each such run is used to observe different parts of the program state, typically using print operations. Doing so allows the first time an incorrect state occurs to be found, and can subsequently be repeated to find the original defect. This works well for deterministic sequential programs (i.e. no I/O etc.), since there is one possible execution path that must be followed each time. Parallel programs however do not share this property, as the random interleaving of two or more programs can produce several distinct execution paths. Interaction with shared memory by parallel programs may lead to *races*, where the components of a parallel compete to update shared memory locations.

As a result, it becomes much harder to reproduce the original failure, and introduces the potential for software bugs that appear and disappear among different execution paths (*Heisenbugs* [28]), *deadlocks* and *atomicity/order violations*.

With cyclic debugging not being suitable for parallel programs, other approaches have been developed. *Record replay debuggers* serialize a specific execution, and use this to force future runs to behave identically, meaning bugs can be reproduced [16]. *Reversible debugging* is described by Engblom [3], and is another alternative that has the ability to step backwards over an execution experiencing an error [1]. This avoids the issue of reproducing an error as no re-execution is required. Some approaches use forward execution from checkpoints to simulate moving backwards [4], while others, such as the Reverse C Compiler (RCC) [17] and our approach presented here, produce an inverted program that executes forwards but simulates reversal. Such approaches will typically reverse an execution in *backtracking* order, where steps of the execution are undone in exactly the inverted order of the forwards execution. Recently, some proposed solutions use *causal-consistent* reversibility [2,6,13], where a step of an execution can be reversed provided all steps that causally depend on it (consequences) have already been reversed. A recently proposed implementation of a causal-consistent reversible debugger is CauDEr [14].

We propose an approach to state-saving reversibility of imperative parallel programs, similar to RCC [17] and both the Backstroke framework [25] and works on it by Schordan [21,22]. We build on our previous work [10,11], and here discuss its application to debugging. We outline this proposal, beginning with the language that we support. We define the process of generating two versions of our original program, the *annotated version* that performs the forwards execution and the state-saving of all required information, and the *inverted version* that uses this saved data to simulate reverse execution. We describe a collection of environments representing our program state, and refer to three sets of small-step operational semantics defined previously [11].

Results that prove our approach to be correct are shown. Our first result shows that the process of state-saving does not alter the behaviour of the original program, as the final program state is unchanged. The second result states that given the inverted version starts in the final program state produced via the annotated execution, execution of this inverted version restores the program state to exactly as it was initially. This result is extended here to hold for all programs, including parallel composition and the challenges it introduces. Our results prove we achieve our aim of implementing correct reversal.

Three examples of common bug types are used to discuss the application of this state-saving reversibility to debugging. Each type of execution is defined in terms of small-step operational semantics, allowing us to advance through an execution one step at a time. This is highly desirable for debugging as intermediate program states can be viewed, allowing the initial effects of a bug to be seen. It also means that bugs leading to crashes can be viewed up to the point of the fatal error, as all previous small steps will have been performed. Similarly for the inverted version, the small-step semantics allow us to return to any intermediate position of our execution. We can also make use of the information saved during the forwards execution prior to its completion. For example, all values any variable has held up to this point will be saved.

We also introduce a simulation tool that implements our state-saving reversibility, specifically the three sets of small-step operational semantics referred to above. This allows the simulator to read and parse an original program, produce the two versions and perform all three types of execution. With the application to debugging being a consideration from the beginning of the development, we then discuss some of the key features that aid debugging. Such features include the ability to force a specific interleaving and the record mode.

Finally, we use this simulator to evaluate the performance of our approach to reversibility. We compare the execution times of programs with and without state-saving, producing an average overhead incurred. We likewise compare the execution times of forwards execution with that of the inverse execution, measuring the performance of reversal. Our main contributions are:

1. An overview of an approach for state-saving reversibility of imperative parallel programs proposed in [11]. A proof showing this holds for all valid programs of our language, extended here to include parallel composition.
2. The application of this method to debugging, explained using three examples of common bugs.
3. A simulator implementing our small-step semantics behind this method and how this is used for debugging.
4. The evaluation of the performance of our approach. This shows an acceptable overhead associated with both state-saving and inversion.

2 Our Approach

We begin with a state-saving approach to reversible execution of imperative parallel programs. A more in depth definition of this approach is available in our previous works [10,11]. Our discussion of this approach is split into the following five broad stages, each of which will be described below.

1. **Language and State.** We extend a typical while language with blocks, local variables, recursion-supported procedures (with no arguments) and parallel composition. We use 'parallel' in this context, but note that we could have used 'concurrent'. We introduce *construct identifiers* and *paths*, necessary to handle local variables and different scopes. The program state is represented as a collection of environments, each of which will be described later.
2. **Annotation.** This process introduces *identifier stacks* into the language syntax, necessary to record a particular run of the program. This produces an annotated version, that when executed saves *reversal information* required for inversion and captures the *non-deterministic interleaving order* via identifiers. This records the outcome of all races introduced by parallelism. All reversal information is stored within the *auxiliary data store.*

3. **Inversion.** This produces the corresponding inverted version of our original program, which itself is a forwards program capable of simulating the reverse execution. It is generated from the executed annotated version.
4. **Running Inverted Program.** Execution of the inverted version will then use both identifiers and the reversal information to simulate the undoing of the execution of the original program.
5. **Debugging.** The ability to execute step-by-step through the inverted version allows us to view the program state at any point. This can be used to compare the expected and actual program state, potentially helping us to find bugs.

Stage 1: Language and State. We begin with a typical imperative while language consisting of assignments, conditional statements and while loops. We extend this with blocks, local variables, procedures (with no arguments) capable of recursion, removal statements and the parallel composition operator `par`. This operator interleaves the execution of two (or more) programs randomly, while removal statements remove local variables or procedures at the end of a block. We refer informally to each argument program of a parallel statement as a *thread*.

Further to this, we also introduce *construct identifiers* and *paths*. Each conditional, loop, block, procedure declaration and procedure call is given a unique name, termed a construct identifier and represented as `In`, `Wn`, `Bn`, `Pn` and `Cn` respectively. These names are of the form `Unique name:Version number`. Each statement that requires evaluation will also contain a *path*, represented as `pa`. This is a sequence of the unique block names in which the specific statement resides, with λ representing an empty path (global). The syntax of this language is shown below, with paths and construct identifiers underlined here only to highlight them, and will be used henceforth without underlining. An example is shown in Fig. 1a, containing two assignments and a while loop performing six iterations. All paths are omitted as there are no blocks meaning all would be λ.

$$\begin{aligned}
\texttt{P} &::= \varepsilon \mid \texttt{S} \mid \texttt{P; P} \mid \texttt{par \{ P \} \{ P \}}\\
\texttt{S} &::= \texttt{skip} \mid \texttt{X = E } \underline{\texttt{pa}} \mid \texttt{if } \underline{\texttt{In}} \texttt{ B then P else Q end } \underline{\texttt{pa}} \mid\\
&\quad\; \texttt{while } \underline{\texttt{Wn}} \texttt{ B do P end } \underline{\texttt{pa}} \mid \texttt{begin } \underline{\texttt{Bn}} \texttt{ DV DP P RP RV end} \mid\\
&\quad\; \texttt{call } \underline{\texttt{Cn}} \texttt{ n } \underline{\texttt{pa}} \mid \texttt{runc } \underline{\texttt{Cn}} \texttt{ P end}
\end{aligned}$$

$$\begin{array}{ll}
\texttt{DV} ::= \varepsilon \mid \texttt{var X = v } \underline{\texttt{pa}}\texttt{; DV} & \texttt{DP} ::= \varepsilon \mid \texttt{proc } \underline{\texttt{Pn}} \texttt{ n is P end } \underline{\texttt{pa}}\texttt{; DP}\\
\texttt{RV} ::= \varepsilon \mid \texttt{remove X = v } \underline{\texttt{pa}}\texttt{; RV} & \texttt{RP} ::= \varepsilon \mid \texttt{remove } \underline{\texttt{Pn}} \texttt{ n is P end } \underline{\texttt{pa}}\texttt{; RP}\\
\texttt{E} ::= \texttt{Var} \mid \texttt{n} \mid \texttt{(E)} \mid \texttt{E Op E} & \texttt{B} ::= \texttt{T} \mid \texttt{F} \mid \neg\texttt{B} \mid \texttt{(B)} \mid \texttt{E == E} \mid \texttt{E > E} \mid \texttt{B} \wedge \texttt{B}
\end{array}$$

The program state is represented as a collection of environments. Firstly, the *data store* σ maps memory locations (**Loc**) to the value (**Num**) they currently hold ($\sigma : (\mathbf{Loc} \mapsto \mathbf{Num})$). Next, the *variable environment* γ maps variables (**V**) to memory locations. Before defining the variable environment, we note that the use of blocks mean that variables can be either global or local, and that a global variable can share its name with multiple local versions. Each such local version

```
X = 5;
N = 0;
while w1.0 (X >= N) do
  X = X - 1;
  Y = Y + 1;
end;
```

(a) Original Loop Program

```
X = 5 [];
N = 0 [];
while w1.0 (X >= N) do
  X = X - 1 [];
  Y = Y + 1 [];
end [];
```

(b) Annotated Loop Program

Fig. 1. Small while loop example: forwards

will have been declared within a different scope, and specifically a different block. This means the unique block name in which a variable is declared is used within γ to differentiate multiple versions ($\gamma : (\mathbf{V} \times \mathbf{Bn}) \mapsto \mathbf{Loc}$). As a result of this, paths are used during variable evaluation to determine the block in which the variable was defined, with this then being used to access the correct memory location. The introduction of parallel composition and local variables mean that *data races* can occur, where the order in which two (or more) steps are performed directly affects the outcome. For example, two assignments to the same variable racing means the assignment performed last produces the final value.

Should the same code be executed in parallel, this approach to distinguishing versions of variables will not be sufficient. For example, consider two procedure calls to the same function on each side of a parallel that both declare a variable using the shared block name (as the same code is being used). In this case, both calls would use the same version of the declared variable, violating correct behaviour. Therefore any reused code, namely procedure and loop bodies, must be *renamed* prior to execution. Explained in [11], procedure bodies are renamed with all construct identifiers updated to begin with the unique call name (that will be different across a parallel). For example, a while loop `w1.0` within a procedure call `c1.0` becomes `c1.0:w1.0`. Loop bodies are renamed with all construct identifiers updated to their next version number. For example, the conditional `i1.0` used in Fig. 5a will become `i1.1` for the first iteration of the while loop. The renamed copies of procedure and loop bodies are stored within the *procedure environment* $\mu : (\mathbf{Pn} \cup \mathbf{Cn}) \mapsto (\mathbf{n} \times \mathbf{P})$ and *while environment* $\beta : \mathbf{Wn} \mapsto \mathbf{P}$ respectively. The *auxiliary data store* is discussed later.

The (forwards-only) execution of programs written in our language is defined in terms of a small-step operational semantics. We do not include this here as it is available in [11]. From here, we refer to each small step as a *transition* (or step) and consider an *execution* to be a sequence of transitions (or steps).

Stage 2: Annotation. Similarly to the Reverse C Compiler (RCC) [17], we produce two versions of an original program. The process of annotation produces the first of our versions, specifically the *annotated version* capable of recording the specific execution. This is implemented via the function *ann*(), shown in [11]. Recording a run of a program can be split into two main tasks, namely

1. Recording required data lost during forward execution (reversal information)

2. Capturing the non-deterministic execution order (due to having `par`).

Firstly, all reversal information is saved during the execution of the annotated program via the operational semantics [11]. This matches closely with the semantics of forwards-only execution, differing only on the state-saving. Such examples of this information include old values overwritten (and lost) as a result of destructive assignments, a boolean value indicating which branch of each conditional was executed, and a sequence of boolean values capturing the number of iterations of each loop. Further, the final value held by a local variable, and any identifiers assigned to a loop/procedure copy (*annotation information*), are saved prior to their deletion via removal statements.

This information is saved in the auxiliary data store δ, keeping all reversal information separate to the program state. This is a collection of stacks, with one for each variable name. All versions of a variable name use a single stack, storing overwritten or final values they held. Using a single stack helps to determine the outcome of races. There is a single stack `B` that holds boolean values for conditionals, and similarly `W` for loops. Finally, the stacks `WI` and `Pr` hold annotation information from loop or procedure body copies prior to removal.

Secondly, the non-deterministic execution order is captured via the use of identifiers. Sequential programs have a single path that can be followed in both directions. Parallel programs have many possible paths, with correct inversion dependent on following the correct inverted path of execution. Not doing so can lead to a state that was not reachable during forwards execution. To avoid this, as each statement of a program is executed, the next available identifier (used in ascending order) is assigned to that statement. In doing so, the overall statement order (interleaving) is recorded as required to ensure correct reversal. The syntax of each statement that requires identifiers to be saved will therefore have a stack for these, represented using `A`. Each identifier is also used to index any reversal information saved for that statement in δ, with all stacks on δ consisting of pairs. Within our operational semantics, any transition that uses an identifier is labelled with it, while those that do not are unlabelled and referred to as *skip steps*. The three types of skip steps are the removal of skip statements as a result of sequential or parallel composition and the closure of a block or loop iteration. The following is the updated syntax for annotated programs, where `P` and `S` are now used to represent annotated programs and statements respectively, and our additional stacks are highlighted via underlining. As with paths and construct identifiers before, these stacks will not be underlined from this point. We omit program expression definitions as they are unchanged.

$$
\begin{array}{ll}
\texttt{S} ::= \texttt{skip I} \mid \texttt{X = E (pa,}\underline{\texttt{A}}\texttt{)} \mid \texttt{if In B then P else Q end (pa,}\underline{\texttt{A}}\texttt{)} \mid & \\
\quad \texttt{while Wn B do P end (pa,}\underline{\texttt{A}}\texttt{)} \mid \texttt{begin Bn DV DP P RP RV end} \mid & \\
\quad \texttt{call Cn n (pa,}\underline{\texttt{A}}\texttt{)} \mid \texttt{runc Cn P end} & \\
\texttt{DV} ::= \varepsilon \mid \texttt{var X = v (pa,}\underline{\texttt{A}}\texttt{); DV} & \texttt{DP} ::= \varepsilon \mid \texttt{proc Pn n is P end (pa,}\underline{\texttt{A}}\texttt{); DP} \\
\texttt{RV} ::= \varepsilon \mid \texttt{remove X = v (pa,}\underline{\texttt{A}}\texttt{); RV} & \texttt{RP} ::= \varepsilon \mid \texttt{remove Pn n is P end (pa,}\underline{\texttt{A}}\texttt{); RP}
\end{array}
$$

As shown in the syntax above, the only difference between an original program and the corresponding annotated version is the presence of identifier stacks within certain statements. Returning to our while loop example shown in Fig. 1a, the corresponding annotated version is shown in Fig. 1b. Execution of the annotated program will populate these identifier stacks with identifiers capturing the execution order. The corresponding executed annotated version containing populated identifier stacks is shown in Fig. 2a.

```
X = 5 [0];
N = 0 [1];
while w1.0 (X >= N) do
  X = X - 1 [3,6,9,12,15,18];
  Y = Y + 1 [4,7,10,13,16,19];
end [2,5,8,11,14,17,20];
```

(a) Executed Annotated Program

```
while w1.0 (X >= N) do
  Y = Y + 1 [4,7,10,13,16,19];
  X = X - 1 [3,6,9,12,15,18];
end [2,5,8,11,14,17,20];
N = 0 [1];
X = 5 [0];
```

(b) Inverted Loop Program

Fig. 2. Small while loop example: inversion

Stage 3: Inversion. After defining annotated execution, the next step is to produce the inverted version via the function *inv*() and execute it via our small-step operational semantics [11]. This version executes forwards as expected, and is produced based on the executed version of the annotated program, meaning all stacks are populated appropriately. The overall statement order is inverted, as well as each declaration statement becoming an equivalent removal statement and vice versa. We use the same syntax for both the annotated and inverted versions, but with `P` and `S` for inverted programs and statements respectively. Returning to our small while loop example discussed throughout previous stages, the inverted version of this program is shown in Fig. 2b. The difference between this and the executed annotated version is the statement order is inverted.

Stage 4: Inverse Execution. Starting in the final program state produced via annotated forwards execution, the inverted version will restore the program state to as it was prior to forwards execution (see results below). The order in which the program executes is determined by the identifiers associated with its inverted statements, with only the statement that has the highest identifier eligible to be executed next. This means we follow backtracking order, where statements are undone in exactly the inverted order of the forwards execution. Backtracking order is necessary to ensure races are reversed correctly. There is however potential for limited causal consistent reversibility, where skip steps and block closures can be executed in any order. A small example of how identifiers capture the execution order and can be used for inversion is shown in [10].

When the choice of the next statement to invert has been made, the reversal information and identifiers saved via annotation are then used to undo the effects of that statement. Specifically this includes the old value of a variable to be restored during a destructive assignment, and boolean values to govern inverse control flow of conditionals or loops. The final value held by local variables will have been saved prior to its removal and so is used to initialise the inverted version, as well as the annotation information that must be used to repopulate all stacks within copies of reused code.

Stage 5: Debugging. In Sect. 4, we will discuss using our approach for debugging and present three examples of identifying common bug types. In Sect. 5, we introduce a simulation tool implementing the process described above, and show how its abilities further aid debugging.

3 Correctness of Our Approach

One motivation for this work is to have an approach to reversible execution of imperative parallel programs that is proved to be correct. Therefore we have proved two properties related to our approach. Prior to discussing these properties, we first provide several definitions and explain important notation.

We begin with defining equivalence. Firstly, two states $\square = \{\sigma,\gamma,\mu,\beta\}$ and $\square'$ $= \{\sigma',\gamma',\mu',\beta'\}$ are equivalent, written $\square \approx \square'$, provided each pair of matching environments are not necessarily identical, but semantically equivalent. Secondly, two auxiliary stores δ and δ' are equivalent, written $\delta \approx_A \delta'$, provided the two stores are semantically equivalent. For example, actual memory locations used within the matching environments may differ, but the 'meaning' is the same. Finally, we define equivalence between a program execution and its corresponding *uniform version*. A uniform execution is a version of an original execution where all skip steps are performed as soon as they are available. Performing skip steps immediately does not alter the behaviour of the program as each such transition does not alter the program state. Therefore a program and its uniform version are equivalent as the program states produced are equal, since the order of transitions using identifiers is unchanged.

We shall use the following notation. A step of forwards only execution is represented using $\hookrightarrow$, while a step of both annotated and inverted execution are represented using $\overset{\circ}{\rightarrow}$ and $\overset{\circ}{\rightsquigarrow}$ respectively, where $\circ$ represents the possible use of an identifier. For example, a destructive assignment is performed to skip via a transition that uses an identifier, while the skip operation is then removed via a transition without an identifier. Uniform versions of both an annotated and inverted execution are represented as $\overset{\circ}{\rightarrow}{}^{*}_{U}$ and $\overset{\circ}{\rightsquigarrow}{}^{*}_{U}$ respectively.

Theorem 1 states an original program and its annotated version behave identically (under the same interleaving) with respect to all environments, except the auxiliary store. This shows annotation has no unwanted side effects.

Theorem 1. *Let* `P` *be an original program,* $\square$ *be the set* $\{\sigma,\gamma,\mu,\beta\}$ *of all environments,* $\square_1$ *be the set* $\{\sigma_1,\gamma_1,\mu_1,\beta_1\}$ *of annotated environments such that* $\square \approx \square_1$ *and* δ *be the auxiliary store. If* $(\texttt{P} \mid \square,\delta) \hookrightarrow^{*} (\texttt{skip} \mid \square',\delta)$*, for some* $\square'$*, then there exists an execution* $(ann(\texttt{P}) \mid \square_1,\delta) \overset{\circ}{\rightarrow}{}^{*} (\texttt{skip I} \mid \square_1',\delta')$*, for some* `I`*,* $\square_1'$ *and* δ'*, such that* $\square' \approx \square_1'$*.*

$$\begin{tikzcd}
(\mathtt{P} \mid \Box) \overset{\circ}{\rightarrow}{}^{*} (\mathtt{skip\ I} \mid \Box') \arrow[r, Rightarrow] \arrow[d, "?"'] & (\mathtt{P} \mid \Box) \overset{\circ}{\rightarrow}{}^{*}_{U} (\mathtt{skip\ I} \mid \Box') \arrow[d, Rightarrow] \\
(\mathtt{P}^{-1} \mid \Box'_1) \overset{\circ}{\rightsquigarrow}{}^{*} (\mathtt{skip\ I'} \mid \Box_1) & (\mathtt{P}^{-1} \mid \Box'_1) \overset{\circ}{\rightsquigarrow}{}^{*}_{U} (\mathtt{skip\ I'} \mid \Box_1) \arrow[l, Rightarrow]
\end{tikzcd}$$

Fig. 3. Diagram representation of proof outline

Theorem 2 states that given an original execution and its annotated equivalent, there exists an inverted execution that starts with the final program state, and restores this to exactly as initially. Shown in [11] to hold for sequential programs only, we note here that it now also holds for parallel programs. As a result, our approach is garbage free, as the auxiliary store is also restored.

Theorem 2. *Let* `P` *be a program and* `AP` *be* $ann(\mathtt{P})$. *Further let* $\Box$ *be the set* $\{\sigma,\gamma,\mu,\beta\}$ *of all environments,* $\Box_1$ *be the set* $\{\sigma_1,\gamma_1,\mu_1,\beta_1\}$ *of annotated environments such that* $\Box \approx \Box_1$, $\Box'_1$ *be the set* $\{\sigma'_1,\gamma'_1,\mu'_1,\beta'_1\}$ *of final annotated environments,* $\Box_2$ *be the set* $\{\sigma_2,\gamma_2,\mu_2,\beta_2\}$ *of inverted environments such that* $\Box_2 \approx \Box'_1$, δ *be the auxiliary store,* δ' *be the final auxiliary store and* δ_2 *be the inverted auxiliary store such that* $\delta_2 \approx_A \delta'$.

If $(\mathtt{P} \mid \Box,\delta) \hookrightarrow^{*} (\mathtt{skip} \mid \Box',\delta)$, *for some* $\Box'$, *and there exists an annotated execution* $(\mathtt{AP} \mid \Box_1,\delta) \overset{\circ}{\rightarrow}{}^{*} (\mathtt{skip\ I} \mid \Box'_1,\delta')$, *for some* `I`, $\Box'_1$ *and* δ', *such that* $\Box' \approx \Box'_1$ *and that the executed annotated version of* `AP` *produced by its execution is* `AP'`, *then there also exists* $(\mathtt{IP} \mid \Box_2,\delta_2) \overset{\circ}{\rightsquigarrow}{}^{*} (\mathtt{skip\ I'} \mid \Box'_2,\delta'_2)$, *for* $\mathtt{IP} = inv(\mathtt{AP'})$ *and some* `I'`, $\Box'_2$ *and* δ'_2, *such that* $\Box'_2 \approx \Box$ *and* $\delta'_2 \approx_A \delta$.

Proof. The diagram shown in Fig. 3 outlines the proof omitted here due to space constraints. From this diagram, we aim to prove the correctness of the arrow labelled with a question mark, and we do so with the three step approach indicated with double arrows. We begin with an arbitrary execution of an annotated program `P` (top left of Fig. 3), and transform this into an equivalent uniform execution (top right of Fig. 3). Recall the definitions of uniform execution and equivalence above. This transformation consists of moving all skip steps (transitions that do not use identifiers) as close to the beginning of the execution as possible, ensuring all dependencies are maintained. An example is a destructive assignment that executes to skip, before this skip is eventually (with other steps potentially interleaved) removed. In a uniform execution, these two steps happen consecutively, with no interleaving of other statements in between.

From this equivalent uniform execution, we then prove two properties by mutual induction on the length of the execution. The first property is similar to that of Theorem 2 and concerns entire executions. This shows that if a uniform annotated program `P` executes to skip, then there exists a uniform inverse execution that, when beginning in the final state equivalent to that produced by forwards execution, also completes producing a program state equivalent to that of prior to the forwards execution. The second property is similar, but concerns only the execution of statements `S`. Since many statements contain complex subprograms, the first property is used by induction here (hence mutual induction). We consider each base case of both properties, and each (mutual) induction case. Using these properties, we obtain the corresponding uniform inverse execution (bottom right of Fig. 3), where $\square_1' \approx \square'$ and $\square \approx \square_1$.

The final step is to relax this uniform inverse execution into a non-uniform equivalent. This process is the opposite of that described for producing a uniform execution, and allows skip steps to be moved appropriately within the execution. Therefore we have shown the arrow from Fig. 3 to be valid, as required.

4 Debugging

This section describes the application of our approach to debugging. Some important aspects of this are:

1. Small-step semantics allow the execution to be paused at any point. Intermediate program states can be viewed, and compared with the expected state. This includes current position and current values of variables.
2. All reversal information saved up to a specific point can also be viewed. This can display the current number of loop iterations, all previous values of a variable and all results of evaluating conditional statements.
3. Program state is accessible in intermediate states, and can be changed to test things including temporary bug fixes.
4. Inversion can be started at any point, allowing debugging of fatal errors.

We now discuss three examples of common bug types, and how our approach to reversibility can be used to aid the process of identifying the underlying cause. We omit all paths and programs within procedure removal statements from all examples, all of which can easily be read from the remaining code.

4.1 Incorrect Logic Bug

We first consider a logic error, typically made by inexperienced programmers. The program in Fig. 2a is intended to have five iterations, however this specific run performs six (as `Y = 6` after execution). The inverted program is shown in Fig. 2b. Beginning in the final program state, the inverted program can be executed forwards for four steps. This involves opening the loop (identifier 20), inverting the final iteration of the loop (identifiers 19 and 18) and finally inverting the second to last condition evaluation (identifier 17). This state, shown in Fig. 2b where all underlined identifiers have been removed and the arrow $\Longleftarrow$ indicates the current position, is now identical to that of the second to last time the condition was evaluated during forwards execution. Using the current program state, we then see that the condition `0 >= 0` holds true, when we expected false. We can see the logic is incorrect, and that replacing the logic symbol within the condition with `>` fixes this bug.

4.2 Parallel - Slow Write

Our second example is of an atomicity violation bug. With a write operation often being slower than a read, we use the program shown in Fig. 4a to simulate this. This contains a race between a read and write of the same variable in parallel. In order to mimic the write operation being slow but atomic, our write is implemented via the procedure `update` that actually performs two assignments, which we assume are performed one after another (with no statements interleaved). This is like saying the write is both slow and atomic. This means the execution will produce one of two possible outcomes. Firstly, the read (line 8) is followed by the write (line 9), meaning `result = 10` and `X = 12` (Outcome 1). Secondly, the write (line 9) is followed by the read (line 8), meaning `result = 12` and `X = 12` (Outcome 2). However, the interleaving shown in Fig. 4a produces an incorrect third state, where `result = 11` and `X = 12`.

The inverted version of our program is shown in Fig. 4b (recall that this is a normal, forwards executing program). Beginning in our incorrect final state described above, the inverse execution first opens the block, re-declares the local variable `X` to the value `12` retrieved from the stack (line 4 using identifier 7), and then re-declares the procedure `update` (lines 5–8 using identifier 6). Next, the parallel statement starts by beginning the inverted procedure call (line 11 using identifier 5). This implies that the write finished last during forwards execution, meaning we should have expected Outcome 1. Then the inverse execution performs the destructive assignment (line 6 using identifier 4). At this point, the only available step is to undo the read now (line 10 using identifier 3). This state, shown in Fig. 4b with all underlined identifiers having been removed, shows that interleaving has occurred, with the arrows indicating current options (at this point in the forwards execution). From this, we observe that interleaving has occurred that directly conflicts our atomicity assumption. Further to this, if we were to continue the inverse execution we would complete the procedure call last, implying the write happened first during forwards execution meaning we should have seen Outcome 2. This inconsistency and the interleaving shown reassures us that we have found the bug. Such a bug can now be fixed, for example, by using an `atomic` construct (which can be easily added to our language).

4.3 Parallel - Race - Airline Example

Our final example is a program implementing a model of an airline that sells tickets via two agents. Each agent remains open and able to sell tickets until there are no remaining free seats. This program is shown in Fig. 5a, where the number of initially free seats is 3, and the number of agents is 2, in order to keep the execution and accompanying environments concise enough for discussion here. We return later to this example and increase both of these when evaluating the

```
begin b1.0
  var X = 10 [0];
  proc p1.0 update is
    X = X + 1 [2];
    X = X + 1 [4];
  end [1];

  par { result = X [3]; }
      { call c1.0 update [5]; }
  remove proc p1.0 update end [6];
  remove var X = 10 [7];
end
//Finishes with result = 11 and
//X = 12
```

(a) Executed Annotated Program

```
//Initial value of result should
//be 12 or 10
begin b1
  var X = 10 [7];
  proc p1.0 update is
    X = (X + 1) [4];
    X = (X + 1) [2];    ⟸
  end [6];

  par { result = X [3]; ⟸ }
      { call c1.0 update [5]; }
  remove proc p1.0 update end [1];
  remove var X = 10 [0];
end
```

(b) Inverted Program

Fig. 4. Slow write example

performance. The specific execution captured in Fig. 5a incorrectly results in 4 tickets being sold, as the final number of free seats is `-1` (`seats = -1`).

The inverted version of this program is shown in Fig. 5b. Beginning in the incorrect final state, the inverse execution will begin by opening the block and re-declaring the local variables and the procedure. Next, the parallel statement is started, with each while loop executing an entire iteration (to simulate the inversion of the closure of each agent) using identifiers 33–24. From here, we now begin the inversion of the penultimate iterations of each while loop. The identifiers 23–14 are used to govern the interleaving across the two threads. The state reached is shown in Fig. 5b where all underlined identifiers have been removed, with the arrows indicating the current position. As this shows, the choice of next step is between the closing of two inverse conditionals. Closing an inverse conditional will reverse the opening of the forwards version, implying that both were open (during the forwards execution) at the same time (consecutive identifiers). Considering each conditional statement as the *critical section* of each thread, we see the mutual exclusion of these sections has been violated. Crucially, when there is a single seat left, if each conditional statement is evaluated consecutively, both conditions will be true. From here, the two calls from each of the true branches will be executed, allocating two seats when only one remains free. Therefore we see there is a race between the read of (conditional evaluation) and write (line 6) to the shared variable `seats`. One solution is to implement the mutual exclusion of the critical sections of each thread (agent).

5 Evaluation of Our Approach

An important next step of our work is to evaluate the performance of this approach. Prior to evaluation, we note that our focus so far has been on proving

```
seats = 3 [0];
begin b1.0
 var agent1 = 1 [1];
 var agent2 = 1 [2];
 proc p1.0 sell is
  seats = seats - 1 [6,11,18,19];
 end [3];

 par {
  while w1.0 (agent1 == 1) do
   if i1.0 (seats > 0) then
    call c1.0 sell [7,20];
   else
    agent1 = 0 [27];
   end [5,8,16,22,26,28];
  end [4,15,25,29];
 } {
  while w2.0 (agent2 == 1) do
   if i2.0 (seats > 0) then
    call c2.0 sell [12,21];
   else
    agent2 = 0 [31];
   end [10,13,17,23,30,32];
  end [9,14,24,33];
 }
 remove proc p1.0 sell end [34];
 remove var agent2 = 1 [35];
 remove var agent1 = 1 [36];
end
//Finishes with seats = -1
```

(a) Executed Annotated Program

```
//Expect seats = 0, not seats = -1
begin b1.0
 var agent1 = 1 [36];
 var agent2 = 1 [35];
 proc p1.0 sell is
  seats = seats - 1 [6,11,18,19];
 end [34];

 par {
  while w1.0 (agent1 == 1) do
   if i1.0 (seats > 0) then
    call c1.0 sell [7,20];
   else
    agent1 = 0 [27];
   end [5,8,16,22,26,28];     ⟸
  end [4,15,25,29];
 } {
  while w2.0 (agent2 == 1) do
   if i2.0 (seats > 0) then
    call c2.0 sell [12,21];
   else
    agent2 = 0 [31];
   end [10,13,17,23,30,32];     ⟸
  end [9,14,24,33];
 }
 remove proc p1.0 sell end [3];
 remove var agent2 = 1 [2];
 remove var agent1 = 1 [1];
end
seats = 3 [0];
```

(b) Inverted Program

Fig. 5. Airline example

this approach to be correct. Identifiers are saved into stacks contained within the syntax, and all reversal information is contained within the additional stacks. Multiple stacks are used as this separation aids the proof, while not necessarily being the most efficient approach. Therefore we remark that all results displayed within this section are produced without any optimization techniques applied.

To aid evaluation, a simulation tool implementing our approach has been developed. An overview and description of key features is shown below. This is used to examine the performance of two keys aspects, namely the overheads or reductions associated with both annotation and inversion.

5.1 Simulation Tool

We have developed a simulator that implements the small-step semantics of our approach [11]. It is capable of reading an original program written in our language from a text file, and parsing this into a linked list structure. This structure can be analysed and used to correctly initialise all of the required environments (including global variables).

The simulator has the ability to simulate all three possible executions, namely traditional forwards only with no state-saving, annotated forwards with state-saving, and inverse. All three executions can be either step-by-step or from start-to-finish. The current program state is viewable at each stage. Annotation and inversion are implemented, transforming an original program into the corresponding annotated and inverse version respectively. The execution of the inverse version follows backtracking in the majority of cases (as discussed above), while also supporting a limited form of causal-consistent reversibility.

The interface of the simulator is currently through the command line. A more user-friendly, graphical user interface (GUI) is currently under development. The following are some of the key features of the simulator.

Auto-generation of Modified Syntax. In order to remove the burden on the programmer, some of the additional parts of the syntax can be automatically generated. This includes the insertion of all unique construct identifiers, paths and removal statements at the end of blocks.

Random or User-defined Interleaving. Any interleaving of programs can either be determined randomly (via random number generation) or by the user at runtime, allowing testing of unlikely executions. This can be switched on/off at runtime, allowing a user to only determine the parts they require.

Record Mode. History logs can be recorded. Firstly, the entire sequence of small-step transitions can be saved. Secondly, for each interleaving decision, all possible choices and an indication of which was chosen can be saved.

5.2 Evaluation

In this section, we consider the following two aspects of our approach.

1. Costs/overheads associated with annotation and state-saving (**Annotation**)
2. Costs/benefits associated with inversion (no evaluation etc.) (**Inversion**)

Evaluation of these aspects consists of timing the executions of three programs written in our language. An average execution time is computed from 100 runs for two execution lengths (e.g. more loop iterations). One aim is to show that any overhead is consistent and does not increase exponentially. All experiments were ran on an Intel Core i5 quad core 3.2 GHz computer with 7.7 Gb memory, running Linux Ubuntu 16.04. Table 1 shows our results, with all times in seconds.

Annotation. Firstly, we consider while loops. The programs **Loop 1** and **Loop 2** (see Appendix A) each contain a while loop with 100 iterations, and a nested while loop with 1,000 and 10,000 iterations respectively. Each of these loops contain a single destructive assignment, meaning 100,000 (Loop 1) and 1,000,000 (Loop 2) of these are performed. Table 1 shows the average overhead introduced as a result of state-saving is 8.3% (Loop 1) and 7.9% (Loop 2).

Next we return to our airline example in Fig. 5a, and extend it with multiple agents (see Appendix B). The programs **Airline 1** and **Airline 2** each have 1000 initially free seats, and contain three and four agents respectively. Table 1 shows the average overhead is 4.6% (Airline 1) and 4.2% (Airline 2).

Finally, we consider all constructs of our language. The programs **General 1** and **General 2** (see Appendix C) each contain two while loops in parallel with 25 and 50 iterations respectively. Each loop contains an assignment and a procedure call, which uses a conditional statement to determine 5 recursive calls. Table 1 shows the average overhead is 13.2% (General 1) and 13.4% (General 2).

Therefore our results show the overhead of annotation for these specific programs to be within the range of 4.2–13.4%. We believe this is reasonable as it does not increase exponentially and given no optimization has been performed. A potential cause of this overhead is the unoptimized process of saving annotation information from copies of loop or procedure bodies prior to the removal of these. Our airline example results also show that increasing the number of programs in parallel does not seem to result in an increased overhead.

Table 1. Performance evaluation of our approach

Program	Original	Annotated	Change from Orig	Inverse	Change from Ann	Change from Orig
Loop 1	0.346	0.375	1.083	0.321	0.855	0.926
Loop 2	3.446	3.717	1.079	3.172	0.853	0.920
Airline 1	0.098	0.103	1.046	0.104	1.013	1.060
Airline 2	0.138	0.144	1.042	0.147	1.019	1.063
General 1	0.033	0.037	1.132	0.037	1.008	1.141
General 2	0.064	0.072	1.134	0.073	1.012	1.147

Inversion. Firstly, we consider the inverse execution time of programs **Loop 1** and **Loop 2**. Table 1 indicates a 7.4% (Loop 1) and 8% (Loop 2) reduction compared to the original execution, and a 14.5% (Loop 1) and 14.7% (Loop 2) reduction compared to the annotated execution.

The inverted executions of the programs **Airline 1** and **Airline 2** are now analysed. Table 1 shows a 6.0% (Airline 1) and 6.3% (Airline 2) increase on the original execution, and a 1.3% (Airline 1) and 1.9% (Airline 2) increase when compared to the annotated execution.

Finally, the programs **General 1** and **General 2** are inverted. Table 1 shows an increase of 14.1% (General 1) and 14.7% (General 2) on the original execution, and 0.8% (General 1) and 1.2% (General) on the annotated execution.

Therefore our results show that for these specific programs running on our unoptimized simulator, the inverse execution time ranges from a 14.7% decrease to a 1.9% increase compared to the annotated execution. A reduction is largely a result of the program containing large amounts of condition/expression evaluation during forwards execution, which is then not required during reversal as appropriate values are retrieved from the auxiliary store. Programs that do not contain large amounts of evaluation may not achieve this reduction, and may be slightly slower. A possible cause is the currently unoptimized process of checking the first identifier of each possible statement to determine the next step.

Though not perfect for comparison since it focuses on Parallel Discrete Event Simulation and distributed systems, the Backstroke framework [25] and work using it by Schordan [21,22] have also been evaluated. In [21], original execution of 100,000 events with a varying number of operations per events was compared to the forwards execution with instrumentation, showing a penalty factor of between 2 and 3 (Mode B). Both the reverse and commit versions are shown to typically be slightly faster than the original execution.

5.3 Related Work

Reversible computation can be applied to Parallel Discrete Event Simulation (PDES) [5], including the Backstroke framework [25] and works by Schordan et al. [21,22]. Backstroke implements a similar approach to that described here, but is capable of handling all of C++ efficiently. To the best of our knowledge, there is no proof of correctness for this framework. Other work focuses on reversible languages, including the imperative languages Janus [26,27], R-CORE [8] and R-WHILE [7], and the object-oriented languages Joule [23] and ROOPL [9]. We employ *identifiers* very much like in the work by Phillips and Ulidowski [18,20]. Causal consistent reversibility of programming languages have been studied, including the recent work on reversible Erlang [14,15], and μOz [6].

6 Conclusion

We have shown a state-saving approach to reversibility of imperative programs containing parallel composition. Our results displayed here prove this method implements correct and garbage free inversion. We have shown there is the possibility of using our approach for debugging, overcoming issues introduced by parallelism, including data races and randomly interleaved execution paths. We have proposed a simulator implementing our reversibility and used it to evaluate the performance. Our experiments show that the overhead incurred as a result of both state-saving and inversion is reasonable. Future work will focus on optimising the simulator, and extending our underlying approach with more constructs to increase the language complexity. We aim to support all constructs of an actual programming language, and potentially to apply our framework to an existing programming language. Extending our limited form of causal-consistent reversibility to allow undoing of more forms of causally independent steps could be also interesting, where we could follow approaches to reversing prime event structures as in [19,24], work on μOz [6], and reversing Erlang as in [14,15].

Acknowledgements. We are grateful to the referees for their detailed and helpful comments. We also thank Shoji Yuen for stimulating discussions on reversing programs for debugging. The authors acknowledge partial support of COST Action IC1405 on Reversible Computation - Extending Horizons of Computing.

A Loop Program

All paths and identifier stacks are omitted as these are automatically inserted by the simulator.

Loop 1. Program used to test performance of while loops

```
X = 100;
while w1.0 (X > 0) do
   Y = 1000;
   while w2.0 (Y > 0) do
      Y = Y - 1;
   end;
   X = X - 1;
end;
```

B Extended Airline

All paths, identifier stacks and removal statements are omitted as these are automatically inserted by the simulator.

Airline 1. Airline model extended with three agents

```
numOfSeats = 1000;
begin b1.0
   var agent1Open = 1;
   var agent2Open = 1;
   var agent3Open = 1;
   proc p1.0 sellTicket is numOfSeats = (numOfSeats - 1); end;
   par {
      par {
         while w1.0 (agent1Open == 1) do
            if i1.0 (numOfSeats > 0) then
               call c1.0 sellTicket;
            else
               agent1Open = 0;
            end
         end;
      } {
         while w3.0 (agent3Open == 1) do
            if i3.0 (numOfSeats > 0) then
               call c3.0 sellTicket;
            else
               agent3Open = 0;
            end
         end;
      }
   } {
      while w2.0 (agent2Open == 1) do
         if i2.0 (numOfSeats > 0) then
            call c2.0 sellTicket;
         else
            agent2Open = 0;
         end
      end;
   }
end
```

C General Program

All paths, identifier stacks and removal statements are omitted as these are automatically inserted by the simulator.

General 1. Program used to test overall performance of our approach

```
begin b1.0
  var left = 25;
  var right = 25;
  var loop1Count = 10;
  var loop2Count = 10;
  proc p1.0 fun1 is
    begin b2.0
      var other = 0;
      if i3.0 (loop1Count > 5) then
        loop1Count = (loop1Count - 1);
        call c1.0 fun1;
      else
        loop1Count = (loop1Count - 1);
        other = other + 1;
      end
    end
  end;
  proc p2.0 fun2 is
    begin b3.0
      var other = 0;
      if i4.0 (loop3Count > 5) then
        loop2Count = (loop2Count - 1);
        call c2.0 fun1;
      else
        loop2Count = (loop2Count - 1);
        other = other + 1;
      end
    end
  end;
  par {
    while w2.0 (left > 0) do
      left = left - 1;
      call c2.0 fun1;
      loop1Count = 10;
    end;
  } {
    while w3.0 (right > 0) do
      right = right - 1;
      call c3.0 fun2;
      loop2Count = 10;
    end;
  }
end
```

References

1. Chen, S., Fuchs, W.K., Chung, J.: Reversible debugging using program instrumentation. IEEE Trans. Softw. Eng. **27**(8), 715–727 (2001). https://doi.org/10.1109/32.940726
2. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_19
3. Engblom, J.: A review of reverse debugging. In: Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference (2012)
4. Feldman, S.I., Brown, C.B.: Igor: a system for program debugging via reversible execution. In: Workshop on Parallel and Distributed Debugging, pp. 112–123. ACM (1988). https://doi.org/10.1145/68210.69226
5. Fujimoto, R.: Parallel discrete event simulation. Commun. ACM **33**(10), 30–53 (1990). https://doi.org/10.1145/84537.84545
6. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 370–384. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_26
7. Glück, R., Yokoyama, T.: A linear-time self-interpreter of a reversible imperative language. Comput. Softw. **33**(3) (2016). https://doi.org/10.11309/jssst.33.3_108
8. Glück, R., Yokoyama, T.: A minimalist's reversible while language. IEICE Trans. **100–D**(5), 1026–1034 (2017). https://doi.org/10.1587/transinf.2016EDP7274
9. Haulund, T.: Design and implementation of a reversible object-oriented programming language. CoRR abs/1707.07845 (2017)
10. Hoey, J., Ulidowski, I., Yuen, S.: Reversing imperative parallel programs. In: Combined Proceedings of EXPRESS/SOS 2017, EPTCS, vol. 255, pp. 51–66 (2017). https://doi.org/10.4204/EPTCS.255.4
11. Hoey, J., Ulidowski, I., Yuen, S.: Reversing parallel programs with blocks and procedures. In: Combined Proceedings of EXPRESS/SOS 2018, EPTCS, vol. 276, pp. 69–86 (2018). https://doi.org/10.4204/EPTCS.276.7
12. Landauer, R.: Irreversibility and heat generation in the computing process. IBM J. Res. Dev. **5**(3), 183–191 (1961). https://doi.org/10.1147/rd.53.0183
13. Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent reversibility. Bull. EATCS **3**, 114 (2014)
14. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: a causal-consistent reversible debugger for Erlang. In: Gallagher, J.P., Sulzmann, M. (eds.) FLOPS 2018. LNCS, vol. 10818, pp. 247–263. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90686-7_16
15. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. J. Log. Algebr. Methods Program. **100**, 71–97 (2018). https://doi.org/10.1016/j.jlamp.2018.06.004
16. LeBlanc, T.J., Mellor-Crummey, J.M.: Debugging parallel programs with instant replay. IEEE Trans. Comput. **36**(4), 471–482 (1987). https://doi.org/10.1109/TC.1987.1676929
17. Perumalla, K.: Introduction to Reversible Computing. CRC Press, Boca Raton (2014)
18. Phillips, I., Ulidowski, I.: Reversing algebraic process calculi. J. Log. Algebr. Methods Program. **73**(1–2), 70–96 (2007). https://doi.org/10.1016/j.jlap.2006.11.002
19. Phillips, I., Ulidowski, I.: Reversibility and asymmetric conflict in event structures. J. Log. Algebr. Methods Program. **84**(6), 781–805 (2015). https://doi.org/10.1016/j.jlamp.2015.07.004

20. Phillips, I., Ulidowski, I., Yuen, S.: A reversible process calculus and the modelling of the ERK signalling pathway. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 218–232. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_18
21. Schordan, M., Jefferson, D., Barnes, P., Oppelstrup, T., Quinlan, D.: Reverse code generation for parallel discrete event simulation. In: Krivine, J., Stefani, J.-B. (eds.) RC 2015. LNCS, vol. 9138, pp. 95–110. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20860-2_6
22. Schordan, M., Oppelstrup, T., Jefferson, D., Barnes, Jr, P.D., Quinlan, D.J.: Automatic generation of reversible C++ code and its performance in a scalable kinetic Monte-Carlo application. In: Proceedings of SIGSIM-PADS 2016, pp. 111–122. ACM (2016). https://doi.org/10.1145/2901378.2901394
23. Schultz, U.P., Axelsen, H.B.: Elements of a reversible object-oriented language. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 153–159. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_10
24. Ulidowski, I., Phillips, I., Yuen, S.: Reversing event structures. New Gener. Comput. **36**(3), 281–306 (2018). https://doi.org/10.1007/s00354-018-0040-8
25. Vulov, G., Hou, C., Vuduc, R.W., Fujimoto, R., Quinlan, D.J., Jefferson, D.R.: The backstroke framework for source level reverse computation applied to parallel discrete event simulation. In: Proceedings of WSC 2011, pp. 2965–2979. IEEE (2011). https://doi.org/10.1109/WSC.2011.6147998
26. Yokoyama, T., Axelsen, H., Glück, R.: Principles of a reversible programming language. In: Proceedings of Computing Frontiers, pp. 43–54. ACM (2008). https://doi.org/10.1145/1366230.1366239
27. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Proceedings of PEPM 2007, pp. 144–153. ACM (2007). https://doi.org/10.1145/1244381.1244404
28. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging, 2nd edn. Academic Press, Cambridge (2009)