Generalized Committed Choice

Joxan Jaffar
School of Computing
National University of Singapore
Republic of Singapore
joxan@comp.nus.edu.sg

Roland H.C. Yap
School of Computing
National University of Singapore
Republic of Singapore
ryap@comp.nus.edu.sg

Kenny Q. Zhu
Microsoft Corporation
Redmond, WA 98052
USA
kennyzhu@microsoft.com

Abstract

We present a generalized committed choice construct for concurrent programs that interact with a shared store. The generalized committed choice (GCC) allows multiple computations from different alternatives to occur concurrently and later commit to one of them. This generalizes the traditional committed choice in Dijkstra's Guarded Command Language beyond *don't care* non-determinism to also handle *don't know* non-determinism. The contribution of the paper is the semantics framework for formalizing GCC. The key challenge is how to handle the notion of commit given that GCC evolves a computation from a single state into many possible co-existing states.

1. Introduction

Nondeterminism means that a computation may need to choose between two or more choices [8]. *Don't care* non-determinism, or *committed choice*, is the most commonly used form of nondeterminism in concurrent programming systems, e.g. Occam [9] and Concurrent Prolog [15]. It's also the basis of many non-deterministic programming constructs such as guarded commands [4], CSP [8], and π-calculus [11]. The original form of committed choice is the *guarded command set* (Dijkstra's guard) [4], written as

$$G_1 \rightarrow S_1 \ [] \ G_2 \rightarrow S_2 \ [] \ \cdots \ [] \ G_n \rightarrow S_n \qquad (1)$$

where G_i is a logical expression, the *guard*, and S_i is a list of statements. The meaning of Dijkstra's guard is that one can choose any S_i to execute so long as its guard G_i is true. Otherwise if all guards are false, then it aborts. Thus the choice gives rise to a form of *don't care* non-determinism because the system doesn't care which S_i to execute so long as its guard is satisfied. This is in contrast with the *don't know* non-determinism used in OR-parallel logic programming [6]. Here, a search space is (non-deterministically) explored to find which choices lead to a solution – the alternatives have to be tried to avoid missing a solution.

In a concurrent programming setting, a meaningful program contains operations that manipulate its environment allowing it to interact with other running programs. A key characteristic of Dijkstra's guard (also guards in Concurrent Logic/Constraint Programming [14]) is that guard G_i is meant as a test to choose an alternative to commit to before performing any other operations which modify the environment. We call this, *early committed choice*.

In this paper, we propose a new choice construct which allows the commit to occur at an arbitrary point within the choice. The following example illustrates our new choice construct, where `cm` denotes a commit:

$$(S_1; \texttt{cm}; S_1') \oplus (S_2; \texttt{cm}; S_2') \oplus \cdots \oplus (S_n; \texttt{cm}; S_n')$$

Dijkstra's guard can then be rewritten in our syntax as

$$(G_1; \texttt{cm}; S_1 \oplus)(G_2; \texttt{cm}; S_2) \oplus \cdots \oplus (G_n; \texttt{cm}; S_n)$$

We call our construct, *generalized committed choice* (GCC), since it generalizes the idea of early committed choice. We assume that all processes are operating in a shared environment in which all variables are global and shared. We call the environment a *store*. The processes don't interact with each other directly, but instead communicate through modifying the values of the variables in the store. The only operations allowed are global variable assignments.

Next we will give a simple example that motivates the development of GCC, and we will present an informal description of the GCC programming model, along with some related work. The remainder of the paper is organized as follows. Section 2 presents a small programming language which we embed GCC. Section 3 introduces the basic runtime structure for GCC. The operational semantics of our simple GCC language is given in section 4. We discuss various possibilities for the meaning of commit in section 5. We argue that the given choice construct is practically implementable in section 6 and propose three key optimization ideas.

1.1 A motivating example

The following example motivates the kinds of non-deterministic choice which are ideal applications for GCC. Imagine two people, among others, participate in an online automated second hand product trading system.

Bob is a photographer who wants to upgrade his equipment. He has two choices: either sell his old camera and buy a better one; or, keep the old camera but sell his old lens and buy a better lens. To avoid ending up with two cameras or selling all his equipment but being unable to buy the upgrade, only one scenario can occur. To maximize buying and selling opportunities, we assume that the buying and selling of items can happen in any order.

Jill wants to downgrade and either sell her good camera or her good lens. Using the proceeds from one of the above sales, she can now buy an average camera.

We can now program the intentions of Bob and Jill as follows. Exclusive choice which is the intent of both Bob and Jill is written as `XOR`. We assume that the market has a clearing function, so that buying an item is only effective if there is a matching sell, and vice versa. Thus both the buy and sell operations are synchronized and block if the corresponding action is not present.

```
Bob:
  (buy(goodlens); sell(averagelens))
                  XOR
  (buy(goodcam); sell(averagecam))
Jill:
  (sell(goodlens); buy(averagecam))
                  XOR
  (sell(goodcam); buy(averagecam))
```

It is easy to see that there is a perfect match between Bob and Jill as Bob can buy Jill's good camera and then sell Jill his average camera. Thus there is a way in which both parties can be satisfied. However, since each party is not directly aware of the other, in this setting, we want them to be able to act independently.

Bob could choose one of the following two non-speculative strategies. The first is for Bob to take a bet on one of the choices and commit to that choice. For example, choose on the first choice that gets to make some progress. That is, if a good lens is for sale, then buy the good lens and take a risk by waiting for a buyer for the average lens. This ignores the second possibility to buy a better camera. Such a "bet-and-risk-it" strategy has obvious pitfalls. What happens if one makes the wrong choice? In the example above, if Bob's program finds a match with Jill's choice of selling good lens first, then the lens (1^{st}) choice of Bob will be committed and the camera (2^{nd}) choice will be eliminated. But as it turns out, Jill wants to buy average camera and not average lens after she has bought a good lens. As such, Bob and Jill are stuck in a deadlock both waiting to complete their trade.

The second and more conservative strategy is for Bob to wait in both of his choices until the conditions for both buying and selling actions are met, and then do both actions atomically. While this is a safer option than the previous one, Bob will certainly miss the trading opportunity with Jill as Jill will not buy his average camera until she has sold her good camera. Both parties will be blocked even when there is a potential solution available.

Though our simple example has only two players, in a real life marketplace, much larger dependency cycles involving more parties may exist. These cyclic dependencies can be resolved through the use of late commit but with the above early commit strategies.

1.2 The Generalized Committed Choice Model

The generalized committed choice allows a new strategy which increases the probability of getting a solution. As Bob has two choices, to maximize his chances, he would like to be able to attempt both choices simultaneously and non-deterministically, and choose the one which succeeds. This leads to a form of speculative computation. We achieve this by having the computation in each choice operate in its own independent "world" containing an independent store. So one world does not effect the other. Now, when Jill comes to the system, her program will join the two existing worlds Bob's program has created. Since Jill also has two choices, her program will further split each world it is living in, and this creates four worlds altogether, each of which represents a possible interaction between Bob's and Jill's choices.

Here, everybody is given the full opportunity to complete their actions: while Bob may take Jill's good lens and get stuck in one world since he cannot sell his good lens; he may able to buy Jill's good camera and sell his average camera to her in another world and then eventually complete the transaction. Thus we also need a way of removing unwanted possibilities which represent other worlds and computations. In every world, Bob's and Jill's computation operate in their own independent reality. They can buy and sell items as if there is no speculation.

The model of GCC enables a programming paradigm where a computation can have a number of distinct possibilities. For simplicity, let us say there are two choices, α and β. We assume that the computations operate on a shared store which can change over time due to external events or through actions of a program. In a choice construct, we allow both α and β to proceed concurrently but isolated from each other. At the same time, there are other computations which are also concurrently interacting with this store. After some computation, one of the choices, say α can choose to commit. This has the effect as if the other possibility β never existed. When we have more than one user, all user choices are multiplied to form a number of worlds. The speculation here is that the result of a computation can be a number of worlds.

This paper proposes a programming model for speculation in a new don't-know non-determinism and concurrency context. The central contribution is a new programming construct GCC which generalizes early committed choice. We formalize the complicated semantics of GCC. The main challenge of the semantics is to deal with the notion of commit in the context of multiple worlds. The main implementation challenge is to contain growth in the number of worlds and to handle the issues of large number of program instances executing in these worlds.

1.3 Related work

Transactions A basic database transaction provides *atomicity* and *isolation* [12]. This can be used in the second strategy depicted above, but since it is unlikely all blocking conditions are satisfied all at once, this method will not work for the example above most of the time. Finally, normal relational databases and SQL do not handle non-determinism, so there is no way for Bob or Jill to specify their choices.

Long-Lived Transactions Long-lived transactions such as "Sagas" [5] do away with the isolation property of the transactions and allow partial changes to the database to be visible to other transactions. Also, one level of nested transactions are allowed in Saga. The problem of saga is that the operations can be unsafe, that is, if one process cannot go through, then the whole saga needs to be compensated. But since the processes can interleave, sometimes no compensation operations are possible, and hence the system becomes irreparably inconsistent. The saga model does not handle the above example firstly because it does not provide non-deterministic choice; secondly, even if choices are possible, multiple choices are operating in the same environment, which means once Bob has bought the good lens for example, he will not have the money to pay for good camera, even if it is available. Furthermore, the compensating transactions have to be provided by the user program rather than being resolved by the system.

CCP and Deep Guards Dijkstra's guards and concurrent constraint programming (CCP) [14] techniques such as GHC [17] and Oz [16, 13] use *early committed choice* to handle non-determinism. Early committed choice can be used to implement the *bet-and-risk-it* strategy, but as we have shown, this may lead to a blocked/deadlocked computation. Furthermore, CCP languages require monotonic stores and is thus not applicable in our context. We point out that *deep guards* in CCP are still early committed choice since although they could include non-determinism and computation, a deep guard is *not* allowed to affect the store.

Transaction Logic $\mathcal{TR}$ and especially the later Concurrent Transaction Logic ($\mathcal{CTR}$) [1, 2] are concurrent, rule-based update programming languages for transactions. They combine long-lived transactions and early committed non-determinism in the framework of logic databases such as Datalog. They do not support our camera example for the same reasons given in the discussion on saga and early committed choice.

Composable Memory Transactions The `orElse` construct in [7] provides a handle to state several *alternatives* in memory trans-

actions. The transaction `s1 ‘orElse‘ s2` first runs `s1`; if it blocks (retries), then `s1` is abandoned with no effect, `s2` is run. If `s2` is also blocked (retries), then the whole transaction retries. In a way, the `orElse` construct offers a don't know type of nondeterminism as it attempts the choices one by one until a satisfying one is found. The semantics of `orElse` differs from GCC, in the former the choices are attempted sequentially, while GCC attempts all choices in parallel.

Active Databases While the *Event-Condition-Action* rules in active databases [18, 3] provide some degree of reactivity to the users, it is still early committed choice, i.e. which rule to fire first. In addition, the actions carried out in ECA rules are either simple database read/write operations or user-defined procedure calls. But in all cases, these actions are done atomically and in isolation, hence interleavings are not possible.

2. Programming with GCC

We illustrate the programming paradigm of GCC with the following simple setting. Let there be a number concurrent or parallel programs (processes) interacting with a common runtime system, which provides a global computation environment or a global memory we call a *store*. Without loss of generality, we assume programs do not use any local variables. Synchronization is achieved by use of the common store and the use of a blocking guarded action. For now, we simply assume the store is a piece of shared memory which contains (variable, value) pairs. The main operation on the store are variable assignments which are atomic.

We now present simple programming constructs for programming with GCC, and the compiled form of these constructs. Although, we have chosen a simple setting, it should be clear from this section that GCC can be easily integrated into more complex programming languages and concurrent/parallel systems.

2.1 The stylized language

We introduce the following minimal concurrent language in which we have added GCC. The grammar of a program r in this paper is defined inductively as:

$r ::=$	
`noop`	No operation
$x := v$	atomic assignment
`if` c `then` r_1 `else` r_2	conditional
`while` c `do` r_1	loop
$c \Rightarrow \delta$	guarded atomic action
$r_1; r_2$	sequence
$r_1 \oplus r_2$	GCC
`cm`	commit this choice
`cu`	commit other choice

The first four constructs are rather standard and thus require little explanation. The assignment operation assigns a value v to a global variable x in the store atomically. Boolean condition c is tested in both the conditional, loop and guard constructs. An example of c is $x + y \leq 10$.

Guarded atomic action, or guard in short, is provided to allow for reactive behavior and enable synchronization among the programs. $c \Rightarrow \delta$, blocks until condition c is true w.r.t. the store and then atomically executes δ. δ is an action such as noop, assignment, `cm` and `cu`, all w.r.t. the store.

The choice construct, $r_1 \oplus r_2$, defines two computations r_1 and r_2 which are to be executed speculatively. For simplicity, in this paper, we only deal with binary choices and it is straightforward to extend this to an arbitrary number of alternatives. Both r_1 and r_2 execute with any updates isolated from each other. Nested choices are allowed. Unless otherwise stated, in the remainder of this paper, we refer to generalized committed choice simply as *choice*.

Within a choice, there are two special operations, namely `cm` and `cu`, which stand for "commit me" and "commit you". These can only be used within the scope of a choice and they are referring to the innermost enclosing choice structure. `cm` expresses the intention to *commit* to this branch of a choice and to remove other branch as if it did not exist; `cu` expresses the intention to *terminate* this branch of a choice and commit to the other branch. Notice that `cm` and `cu` are *not* symmetrical since after executing `cm` in a branch, the program can continue, whereas in the case of `cu`; the program instance will be *halted*. If `cm` and `cu` are used outside the scope of a choice, they have no effect. Furthermore, only the first use of `cm` or `cu` has any effect within a choice branch. We require that every choice branch must have a commit operation, the intent is that a choice must eventually be "committed". This can be achieved by systematically adding `cm` to the end of every choice branch.

The reason we do not provide a parallel composition in this language is because this is an open paradigm for parallel programs. The parallel programs (agents) themselves are the parallel composition.

2.2 The abstract assembler language

The above simple language is meant to be compiled into the following abstract assembler language which is executed by the GCC runtime system. Every instruction below takes one step to execute. The exact semantics of these instructions are in Section 4.

- `noop`: does not do any operation but one time step elapses.
- `x := v`: assigns the value v to variable x.
- `cm`: denotes commit this choice branch.
- `cu`: denotes commit the other choice branch and terminate this branch.
- `test(c, pc1, pc2)`: tests condition c, if true go to program counter `pc1`, else go to program counter `pc2`. This is used in both the conditionals and the while loops.
- `goto(pc)`: goes to program counter `pc`. `goto` is used in the while loops and choices.
- `guard(c, δ)`: tests condition c, and if true execute δ in one step, else does nothing. Note that δ is an operation that is either `noop`, `x:=v`, `cm` or `cu`.
- `begin_choice (pc1, pc2)`: denotes the begin of a choice construct and specifies the next program counter for both branches. The left branch goes to `pc1`; and the right branch goes to `pc2`.
- `end_choice`: denotes the end of a choice construct.
- `halt`: halts the current program.

We use the Bob example to illustrate the use of the simple programming language and its compiled form. For simplicity, we have not dealt with the details of the transactions and have only expressed the requirements of the example in terms of availabilities of the items.

Stylized syntax:

$$(goodlens \geq 1 \Rightarrow goodlens := goodlens - 1;\ averagelens := averagelens + 1); cm$$
$$\oplus$$
$$(goodcam \geq 1 \Rightarrow goodcam := goodcam - 1;\ averagecam := averagecam + 1); cm$$

Compiled form in assembler code:

```
<0> begin_choice(<1>, <5>)
```

```
<1>  guard(goodlens>=1, goodlens:=goodlens-1)
<2>  averagelens:=averagelens+1
<3>  cm
<4>  goto(<8>)  //This ends left branch
<5>  guard(goodcam>=1, goodcam:=goodcam-1)
<6>  averagecam:=averagecam+1
<7>  cm
<8> end_choice
<9> halt
```

3. Stores and Worlds

In this section, we formalize a suitable runtime structure for GCC. We begin with a conventional definition of a single store, which is then extended to multi-stores and multi-worlds.

3.1 Stores

Figure 1. A single store Δ

Programs work with and interact with other programs using a store. We will assume that a *store*, denoted by Δ represents a set of variable-value pairs (x, v), for example, $x = 5$, $y = 8$, etc. So the store functions as shared memory, albeit slightly higher level, which a number of concurrent programs can access and modify. As we will be considering more than one store, we will also annotate a store with a version number, i.e. we depict a store graphically as a triangle with a version number in Fig. 1.

We will also make use of intersection and union of stores which is defined as follows:

$$\Delta_1 \cap \Delta_2 = \{(x,\ v) | (x,\ v) \in \Delta_1 \ \wedge (x,\ v) \in \Delta_2\}$$
$$\Delta_1 \cup \Delta_2 = \{(x,\ v) | (x,\ v) \in \Delta_1 \ \vee (x,\ v) \in \Delta_2\}$$

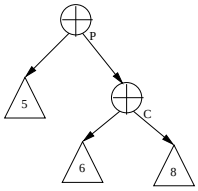


Figure 2. A multi-store

The runtime structure of a program using GCC in general consists of a collection of stores organized in a tree structure. We call this structure (see Fig. 2) a *multi-store*. The leaves of the tree are the stores, and the intermediate nodes are choice nodes $\oplus_{id}$, where id is a unique identifier. The multiple stores are used to represent different possible values for the variables which may have been modified by the alternatives of a choice.

3.2 Continuations and worlds

The operational semantics of the GCC and the program language is centered around the creation, evolution and deletion of a number of worlds. Before we introduce the notion of worlds and multi-worlds, we first define a program *continuation.*

DEFINITION 1 (Continuation). *A program continuation σ is a triple* `<P, pc, cids>`*, where P stands for the program code (the abstract assembler code), pc is the program counter which initially is 0, and $cids$ is a stack of choice ids dynamically generated for each choice construct being processed in this program. Initially $cids$ is the empty stack.*

The runtime system with GCC is characterized as a number of program continuations of executing programs which may be updating the various stores in the multi-store.

DEFINITION 2 (World). *A world is a pair (Δ, Σ), where Σ is a set of continuations of all the programs interacting with Δ.*

In other words, every store in the runtime system is associated with a set of continuations. Programs execute when the system picks one continuation from the set, and advances it by executing an instruction, known as a program step.

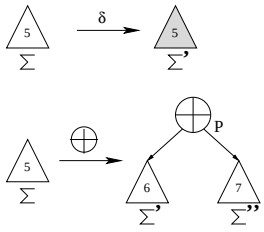


Figure 3. Evolution of a world to a multi-world

Given a world, the execution of one program step by one of the continuations either evolves the world to a new one (the store and continuation change), or it can split the world into two new worlds using a choice with two alternatives, i.e. a choice point. This is illustrated in Fig. 3. The upper transition is the case when a continuation issues an update δ, i.e. x := 10, which changes store Δ_5 to its new state (denoted by the gray color). The lower transition shows the splitting of one world into two, when a continuation σ issues a choice point $\oplus_{id}$, and this creates two new stores, Δ_6 and Δ_7, each of which is a copy of the original store Δ_5, and also two sets of continuations Σ' and Σ'' for each branch in the choice point. We call the tree runtime structure with multiple worlds as a result from executing various choices, a *multi-world*. Later, we will see how `cm` and `cu` help reduce the number of worlds by "chopping" sub-trees from the multi-world.

A multi-world can be formally defined as a tree:

$$\begin{array}{rcll} \mathcal{T} & ::= & \Delta_i, \Sigma & \text{(single store with continuations)} \\ & | & \mathcal{T}_1 \oplus_{id} \mathcal{T}_2 & \text{(tree of stores with continuations)} \end{array}$$

We can think of $\oplus_{id}$ as a logical XOR, and the multi-world is just the union of the a collection of worlds organized in a tree structure. Note that a subtree of multi-worlds is also a multi-world.

4. Operational Semantics

We can now describe the operation semantics of GCC and the simple programming language in terms of transitions on multi-world states. Initially, there is a single world. This can then evolve into a multi-world through the use of choice. The computation can possibly become more determined once commits reduce the alternatives. It may reduce down to a single world again. Each world in a multi-world is independent from each other and evolves separately. In the following semantics, the system will either execute a *program step* (P-step), a *commit step* (C-step), or an *environment step* (E-step),

4.1 P-step

A P-step advances one of the continuations in a world, i.e. a program gets to execute. In what follows, a P-step is applied to a selected world, (Δ, Σ), from the multi-world. A continuation σ is

selected from one of the current executing program continuations Σ and one of the P-step rules are applied. In general, the P-step rules have the following form:

$$\Delta, \sigma \longrightarrow \Delta', \sigma' \quad (2)$$

where the store can possibly be changed to a new store, Δ' through an update from the program and the new continuation is σ'. We use $next(\sigma)$ to denote a continuation which is the same as σ except that the pc refers to the next assembler instruction to follow the one in σ; $\mathcal{I}(\sigma)$ to denote the abstract assembler statement at continuation σ; and $\Delta[x = e]$ to denote a new store where the value of x is replaced by e.

We first look at P-steps which do not involve choices and commits. The rules for the transition $\Delta, \sigma \longrightarrow \Delta', \sigma'$ for each of the following abstract assembly instructions are as follows:

- Assignment: $\mathcal{I}(\sigma) = (x := e)$
  The new store $\Delta' = \Delta[x = e]$, i.e. it is updated with the assignment, and $\sigma' = next(\sigma)$. Note that external variables are read-only and cannot be updated.
- Test: $\mathcal{I}(\sigma) = (test(c,\ pc_1,\ pc_2))$
  The store is unchanged, $\Delta' = \Delta$. If $\Delta \models c$ then σ' is the same as σ except the program counter becomes pc_1; otherwise the program counter in σ' becomes pc_2.
- Goto: $\mathcal{I}(\sigma) = (goto(pc))$
  The store is unchanged, $\Delta' = \Delta$, and σ' is the same as σ except the program counter in σ' becomes pc.
- Noop: $\mathcal{I}(\sigma) = (noop)$
  The store is unchanged, $\Delta' = \Delta$, and $\sigma' = next(\sigma)$.
- Guard: $\mathcal{I}(\sigma) = (guard(c,\ x := e))$
  This executes when $\Delta \models c$ with the new store $\Delta' = \Delta[x = e]$ and $\sigma' = next(\sigma)$. So the guard only executes the $x := e$ when the condition c is true given store Δ and otherwise blocks execution of the instruction. In practice, the blocking and re-enabling of programs requires a *trigger mechanism* [10], which is not the focus of this paper.
- Halt: $\mathcal{I}(\sigma) = (halt)$
  As the program halts, we simply remove its continuation from the world, therefore the transition is $\Delta, \sigma \longrightarrow \Delta$
  The store is unchanged.

The semantics for the above non-choice constructs in the language is straightforward because it deals just with a single world, i.e. one store and the pc part of the continuation. We now turn to the choice and commit constructs which affect the multi-world. From one world, the choice construct creates two possible worlds. Each world begins with the same store and has its own continuation for the computation for each alternative of the choice. In the compiled assembly language form, the end of the scope of a choice is denoted by an `end_choice` instruction.

- `begin_choice`: $\mathcal{I}(\sigma) = (begin\_choice(pc_1,\ pc_2))$
  The transition which creates results in two new worlds is
  $\Delta,\ \sigma \longrightarrow (\Delta,\ \sigma') \oplus_{id} (\Delta, \sigma'')$
  where $\sigma = \langle P, pc, cids\rangle$, id is a fresh identifier for the multi-world choice constructor $\oplus$, $\sigma' = \langle P, pc_1,\ push(id,\ cids)\rangle$, and $\sigma' = \langle P, pc_2, push(id,\ cids)\rangle$. Recall that $cids$ is a stack of choice identifiers in the continuation. Thus, $push$ returns a new stack with id pushed on top.
- `end_choice`: $\mathcal{I}(\sigma) = (end\_choice)$
  As this is executed in a single world, the transition is
  $\Delta,\ \sigma \longrightarrow \Delta,\ \sigma'$
  where the store is unchanged, $\sigma = \langle P, pc, cids\rangle$, and $\sigma' = \langle P, pc', pop(cids)\rangle$. The new pc' refers to the next instruction. Pop returns a new stack minus the topmost element.

Notice that although the `end_choice` instruction is executable in each of the alternative worlds in the choice, they are managed independently since the continuations are separate.

The intent of the commit instructions is to be able to remove some worlds. This is achieved with a combination of a P-step which has the effect of a special store update and a C-step which will delete some (possibly zero) worlds. The commit rules for the transition $\Delta,\ \sigma \longrightarrow \Delta',\ \sigma'$ are as follows and we will also use a top function which returns the topmost element of the stack:

- cm: $\mathcal{I}(\sigma) = (cm)$
  Let $\sigma = \langle P, pc, cids\rangle$. If the the stack $cids$ is empty, then the cm behaves like a noop, so $\Delta' = \Delta$. Otherwise, the cm is in the scope of a choice. Let $id = top(cids)$. The new store $\Delta' = \Delta[cm_{id} = 1]$ records that this alternative has committed. In all cases, $\sigma' = next(\sigma)$. Notice that executing another cm has no effect, thus multiple cm's are idempotent.
- cu: $\mathcal{I}(\sigma) = (cu)$
  Let $\sigma = \langle P, pc, cids\rangle$. If the the stack $cids$ is empty or $(cm_{id},\ 1) \in \Delta$, then the cu behaves like a noop, so $\Delta' = \Delta$ and $\sigma' = next(\sigma)$. The second case above is when a cu occurs after a cm, therefore it has no effect as the choice has already been committed. Otherwise, we want to commit the other branch. Let $id = top(cids)$. The new store $\Delta' = \Delta[cu_{id} = 1]$ records that the other alternative is to be chosen. As cu should now halt, the transition for a cu which commits is
  $\Delta, \sigma \longrightarrow \Delta$
  since its continuation is removed.

Both rules set a flag in the store which signifies that the corresponding choice identifier has committed in a world. Note that cm and cu are not symmetric here.

4.2 C-step

A C-step is used to remove worlds in a multi-world corresponding to the other alternative in a choice which has executed a commit. We now give the rules for *coordinated commit*. Consider a multi-world with a subtree of the form $\mathcal{T}_1 \oplus_{id} \mathcal{T}_2$, the C-step rules are:

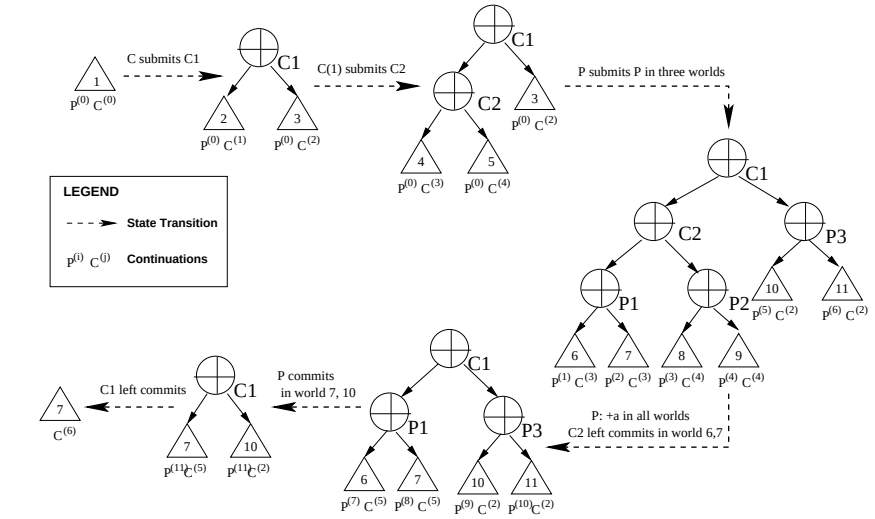
- $\mathcal{T}_1 \oplus_{id} \mathcal{T}_2 \longrightarrow \mathcal{T}_1$ if $\mathcal{CV}(\mathcal{T}_1) \models \mathtt{cm}_{id} = 1$. This removes the worlds in $\mathcal{T}_2$ because the coordinated commit condition means that all the worlds in alternative $\mathcal{T}_2$ have committed to the left choice.
- $\mathcal{T}_1 \oplus_{id} \mathcal{T}_2 \longrightarrow \mathcal{T}_2$ if $\mathcal{CV}(\mathcal{T}_1) \models \mathtt{cu}_{id} = 1$. This is similar to cm except that that the commit is on the opposite alternative of the choice.

Note that the C-step rules are used in conjunction with the P-step rules. In some worlds of the multi-world, commits corresponding to this choice are executed. Coordinated commit then selects some worlds to keep and some to remove from the multi-world based on which commits have been executed in the multi-world choice structure. It is called coordinated commit since only when all the worlds which arise from a choice sub-tree in the multi-world have consistently committed to the same alternative do we allow the worlds in the other alternative to be removed.

Coordinated commit is just one possible semantics for commit. In this section, we have focused on coordinated commit as it is a reasonable choice for commit. A discussion which compares coordinated commit with other alternatives is given in section 5.

4.3 E-step

An E-step occurs when an external variable, let us say x, is changed by the external environment. In an E-step, all occurrences of x in the multi-world are changed to its new value.



Continuations Listing:

$P^{(0)}$: $\langle P,\ 0,\ [] \rangle$ $P^{(1)}$: $\langle P,\ 1,\ [P1] \rangle$ $P^{(2)}$: $\langle P,\ 5,\ [P1] \rangle$ $P^{(3)}$: $\langle P,\ 1,\ [P2] \rangle$
$P^{(4)}$: $\langle P,\ 5,\ [P2] \rangle$ $P^{(5)}$: $\langle P,\ 1,\ [P3] \rangle$ $P^{(6)}$: $\langle P,\ 5,\ [P3] \rangle$ $P^{(7)}$: $\langle P,\ 2,\ [P1] \rangle$
$P^{(8)}$: $\langle P,\ 6,\ [P1] \rangle$ $P^{(9)}$: $\langle P,\ 2,\ [P3] \rangle$ $P^{(10)}$: $\langle P,\ 6,\ [P3] \rangle$ $P^{(11)}$: $\langle P,\ 9,\ [] \rangle$
$C^{(0)}$: $\langle C,\ 0,\ [] \rangle$ $C^{(1)}$: $\langle C,\ 1,\ [C1] \rangle$ $C^{(2)}$: $\langle C,\ 10,\ [C1] \rangle$ $C^{(3)}$: $\langle C,\ 2,\ [C2,\ C1] \rangle$
$C^{(4)}$: $\langle C,\ 5,\ [C2,\ C1] \rangle$ $C^{(5)}$: $\langle C,\ 8,\ [C1] \rangle$ $C^{(6)}$: $\langle C,\ 12,\ [] \rangle$

Figure 4. A worked example

4.4 An example of multi-world transitions

Consider the following producer/consumer example. We use a simplifying notation to increase readability. We write $?x$ to denote a guard that it is possible to consume, and the condition is $(x \geq 1)$. Production is denoted by $+x$, represented as the assignment $x := x + 1$. Consumption is denoted by $-x$, which is $x := x - 1$.

$$P ::= (+a;\ +b;\ \texttt{cm}) \oplus (+a;\ +c;\ \texttt{cm})$$
$$C ::= ((?a \Rightarrow -a;\ \texttt{cm} \oplus_2 ?b \Rightarrow -b;\ \texttt{cm});\ \texttt{cm}) \oplus_1 ((time \geq 10) \Rightarrow \texttt{cm})$$

For convenience of discussion below, we have numbered the choice constructs in program C. $Time$ is an external variable representing an external clock. The program can be translated into assembler as follows:

```
Reactor P:
<0> begin_choice(<1>, <5>)
<1>  a:=a+1
<2>  b:=b+1
<3>  cm
<4>  goto <8>  //end left choice branch
<5>  a:=a+1
<6>  c:=c+1
<7>  cm
<8> end_choice
<9> halt

Reactor C:
<0>  begin_choice(<1>, <10>)
<1>   begin_choice(<2>, <5>)
<2>    guard(a>=1, a:=a-1)
<3>    cm
<4>    goto <7> //end left branch of choice 2
<5>    guard(b>=1, b:=b-1)
<6>    cm
<7>   end_choice
<8>   cm
<9>   goto <11> //end left branch of choice 1
<10>  guard(time>=10, cm)
<11> end_choice
<12> halt
```

We illustrate in Fig. 4 the dynamic evolution of the multi-world with the two concurrent programs P and C where the initial store is Δ_1, namely $\Delta_1 = \{a = 0,\ b = 0,\ c = 0,\ time = 0\}$. For the ease of discussion, we number the choice nodes of P as $P1$, $P2$ and $P3$. We denote the continuations attached to each world as $P^{(i)}$ or $C^{(i)}$ with the details of the program continuations given at the bottom of Fig. 4.

Suppose C gets to make its choices first, the result is three worlds with the two choice notes $C1$ and $C2$ in the multi-world. Then P starts to issue a choice which multiplies to six different worlds. As P produces a and b in one branch and a and c in another, before P commits in any of its branches, choice $C2$'s left branch can consume a and commit. The commit adds the information $\texttt{cm}_{C2} = 1$ to both worlds, hence the C-step can be applied to prune off the worlds of Δ_8 and Δ_9. In the $P3$ subtree, P commits in world 10 first and deletes world 11. P can also commit in the the right branch of the $P1$ subtree, using a C-step to prune off the left subtree of $P1$, namely Δ_6. At this point, there are only two worlds left, Δ_7 and Δ_{10} Since P now has committed in all worlds (7 and 10). Finally, the left branch of choice 2 in C commits which kills world 10 so there is only one remaining world. The speculation within the choices eventuates to a definite result.

5. On the semantics of commit

The key issue of commit is when a `cm` or `cu` is executed in some world, which other worlds should be deleted and when to delete them. This amounts to which parts of the multi-world to prune and when to prune them.

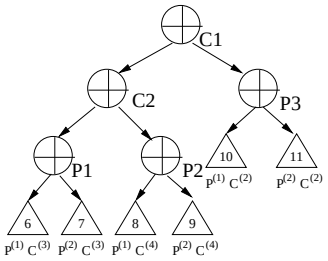


Figure 5. A multi-world

For the purpose of discussion, we extract out a multi-world from Fig. 4 and show it as Fig. 5. As we can see, the choice in P can be duplicated into different parts of the tree, and these choice nodes have different ids, as they are created in different worlds and hence belong to different *scopes*. Subsequently other programs that are interacting with stores Δ_6 through Δ_{11} can also issue choices and further expand the tree. The coordinated commit works as follows. A commit operation not only matches its id syntactically with the choice structure it belongs to, but also maps itself to the scope in which the choice was first launched into. For example, if a commit is executed in one of the worlds under node $P1$ in Fig. 5, then only some of the worlds under $P1$ should be deleted and not worlds under $P2$ or $P3$ in the tree. This is accomplished by the choice id stack, $cids$, in each program continuation. A commit always uses the id of the inner-most choice construct that surrounds this commit operation. And since the ids are generated dynamically as choices are issued, the same choice construct will obtain different ids in different scopes. Thus the matching of commits with the correct choice nodes is done automatically.

We further discuss a few other alternatives for the semantics of commit, and argue why the given semantics in section 4 is selected. We will illustrate these semantics in Fig. 6, 7 and 8. For simplicity, the continuations associated with the worlds are omitted. Where commit has been executed to a store Δ_i, we write $\texttt{cm}_{id}$ under Δ_i. The stores with the commits of interest are highlighted.

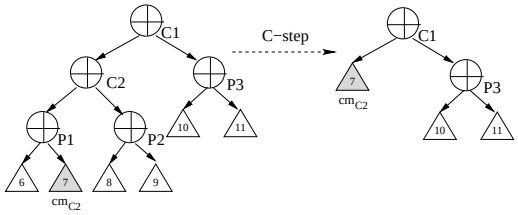


Figure 6. Absolutely eager commit

The first kind of commit is an *absolutely eager* commit. Here, the effect of a commits has no delay. If a `cm` whose id is i is reached in any world, this world is committed and all other worlds within the scope of this `cm`, that is, under the subtree rooted at $\oplus_i$, are killed immediately, leaving just one world under $\oplus_i$. That effectively removed all the intermediate nodes including $\oplus_i$ in that subtree (see Fig. 6). This form of commit does not seem very useful since this it allows for minimal inter-play of choices, and the chances of achieving a useful speculation is small since other possibilities are eliminated immediately. In addition, with this semantics, `cu` does not make sense as `cu` may refer to a choice currently in many worlds and the system does not know which world to commit to.

A "less eager" kind of commit is *eager coordinated* commit. For a program $r ::= r_1 \oplus r_2$, if `cm` is reached in one of the worlds

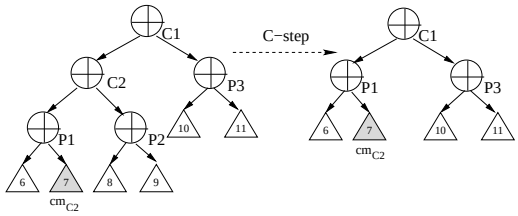


Figure 7. Eager coordinated commit

where r_1 is executed, then all worlds associated with r_2 are deleted immediately (see Fig. 7). The idea here is that syntactically r_2 is not a viable choice any more. Conversely, if `cu` is reached by r_2 in any world, then all worlds associated with r_2 are deleted immediately. However, in either cases, r returns only after r_1 has committed in *all* the remaining worlds. This type of commit is "eager" because commit in one world kills the alternative choice in all other worlds; it is "coordinated" as the program only returns after the choice can commit in all remaining worlds. The following example of two programs P and C shows a drawback of this semantics,

$$P ::= (+a; \texttt{cm}) \oplus (+b; \texttt{cm})$$
$$C ::= (?a \Rightarrow -a; \texttt{cm}) \oplus (?b \Rightarrow -b; \texttt{cm})$$

Four worlds are created with these two programs. We denote the world in which the left branch of P and left branch of C interact as P_lC_l and likewise for other worlds. Assume nothing exists in the store initially, P produces a and b in all four worlds but hasn't committed. Now suppose C consumes a in P_lC_l and commits, which kills worlds P_lC_r and P_rC_r. However, as it turns out, P now commits in world P_rC_l first and kills P_lC_l, which renders C in a blocked state. Had C not killed P_lC_r and P_rC_r early, both P and C may commit in world P_rC_r.

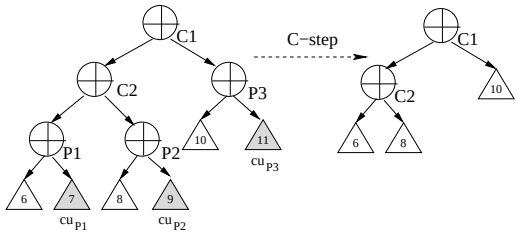


Figure 8. Late coordinated commit

A lazy alternative commit semantics is the *late coordinated* commit. Here the system only start removing worlds when a program $r ::= r_1 \oplus r_2$ with a choice construct has committed its r_1 (or r_2) branch in *all* the worlds with which the r_1 (or r_2) branch continuations are associated. In other words, commits are not only coordinated within a scope, but also across all scopes this program was associated with. For example, in Fig. 8, worlds 7, 9 and 11 are only removed when a C-step `cu` in program P has been executed in all these worlds. Note that worlds 7, 9, 11 are the only worlds where the right branch of P exists. While this semantics increases the possibility of getting a solution and decreases the chances of deadlock from a system point of view, it is unduly conservative in chopping the tree which may cause the multi-world to be too large. At the same time, there is a potential problem when one branch continuation of r cannot proceed to commit due to blocking, no worlds due to r can be removed from the multi-world.

The coordinated commit introduced in Section 4 can be considered as a compromise between the eager coordinated commit and

the late coordinate commit. It does not kill the alternative choice until `cm` of this choice has been reached in all worlds in the scope of the choice construct. This solves the deadlock problem caused by eager coordinated commit, depicted in the example above. At the same time, it does not over-delay the removal of the worlds so that the size of the multi-world can be contained more effectively.

Finally, we remark that it is possible to have a mixed semantics of commit in GCC. Though we have presented one fixed commit semantics in Section 4, it is straightforward to extend it to several possible semantics. In a closed system, concurrently interacting programs can be determined and controlled, hence it makes sense for programs to specify their desired version of commit. In this paper, we consider the more general setting of an open system where agents can dynamically submit arbitrary programs. Here, having multiple commit semantics makes less sense because unpredictable behavior of the system can negate the extra program control associated with the mixed semantics. For example, a late committed choice could be killed by an eager commit issued by another program unexpectedly.

6. Key Implementation Concepts

The speculation in GCC means that the choices in one program are multiplied with choices from all other programs. This means that when choices do not or cannot commit for a long time, the multi-world can grow exponentially large. We remark that this is not a drawback but rather a consequence of allowing the power which comes from committing "late" and the ability to speculate. Furthermore, the amount of parallelism or concurrency can also become exponentially large since the programs are running in parallel at the leaves.

The main challenge to realize GCC is to enable optimizations which can reduce the space and computation requirements. In this section, we discuss some key implementation ideas which address these issues as follows:

1. re-distribution of data in the multi-worlds
2. re-distribution of program continuations in multi-worlds
3. structure-sharing of portions of the multi-world

6.1 Reducing data storage

Earlier we defined conjunctive views on the multi-world. We now define a *differential view*, $\mathcal{DFV}$, for every node of the multi-world except the root node.

DEFINITION 3 (Differential view). *For a subtree $\mathcal{T}_1 \oplus_{id} \mathcal{T}_2$,*

$$\mathcal{DFV}(\mathcal{T}_1) = \mathcal{CV}(\mathcal{T}_1) \setminus \mathcal{CV}(\mathcal{T}_1 \oplus_{id} \mathcal{T}_2)$$
$$\mathcal{DFV}(\mathcal{T}_2) = \mathcal{CV}(\mathcal{T}_2) \setminus \mathcal{CV}(\mathcal{T}_1 \oplus_{id} \mathcal{T}_2)$$

For special case of the root of the multi-world $\mathcal{T}_0$,

$$\mathcal{DFV}(\mathcal{T}_0) = \mathcal{CV}(\mathcal{T}_0)$$

In our original semantics, the store of each world is kept at the leaves of the multi-world tree. This can lead to a lot of data duplication if only a few variables are relevant to speculation. The idea of the optimization is to re-organize the multi-world so that, instead of having the data at the leaves, portions of the stores can be materialized at other nodes in the tree by using a differential view. In essence, this method stores common data as high as possible in the tree, to reduce storage redundancy. This results in the root node storing the common data of all the worlds; its left (right) child stores the common data of all the world in the left (right) sub-tree minus the common data of the root, and so on.

One can use a strategy that periodically materializes the $\mathcal{DFV}$ at respective nodes and thus save storage space. This can make evaluation of multi-world queries more efficient as common data are stored higher and the top-down queries are more likely answered early in the tree traversal.

For efficient evaluation of guard conditions, we can make use of materialized disjunctive view. Guard conditions are much more expensive than tests if the guards block. A blocked guard is only enabled by another update either by another program or by an external variable change. The reason why the disjunctive view is useful at a particular node in the multi-world is that a blocked guard may occur in many worlds in the sub-tree from that node. So a disjunctive view can be used as an indexing condition to approximate whether a change in the view might wake up a blocked guard.

However, disjunctive views are large and can be too expensive to materialize and is counterproductive if one is trying to reduce storage redundancy. The idea then is to store an approximation $\mathcal{CP}$ of that view $\mathcal{DV}$, such that $\mathcal{DV} \models \mathcal{CP}$. We call such approximation $\mathcal{CP}$, a *common property* of a multi-world $\mathcal{T}$, written as $\mathcal{CP}(\mathcal{T})$. There are many choices for a $\mathcal{CP}$, and it is probably best left to the implementor of the runtime system. The simplest $\mathcal{CP}$ of a variable with a set of ordered values, is to use an interval. For example, if a multi-world contains the following data about variable x: $x = \{3,\ -2,\ 5,\ 6,\ 9\}$, a possible common property of that tree for x is $-2 \leq x \leq 9$. So if under this $\mathcal{CP}$ a blocked guard was not satisfied previously, then it would still not be woken up by any updates to the stores for which this approximation is still valid. We remark that $\mathcal{CP}$ is used not only for guards but also for other purposes which we will explain in the next section.

6.2 Reducing the number of continuations

There are two sources of concurrency and parallelism in a multi-world. Firstly, the internal concurrency between possibly interacting programs in one world. Secondly, we may have many worlds which run independently of each other. The idea here is that we can reduce the amount of computation needed by reducing the number of program continuations in the multi-world. To achieve this, we can collect continuations which are identical copies (due to other programs splitting the worlds) from the leaves, treat them as one copy, which we call the *synchronous continuation* and execute it at a higher node in the multi-world tree. In other words, instead of running many copies of the same program on the leaves, we run just one copy in a higher internal node, and thus save computation.

Recall from the last section that we can materialize $\mathcal{DFV}$'s and $\mathcal{CP}$'s in the internal nodes, and the current strategy of "gathering" continuations up works well because it is likely that a synchronous continuation can be moved up to an internal node where the data it operates on are materialized.

Of course, it is not always safe to "synchronize" the continuations from the leaves but there are many important cases when this is possible. For example, suppose there is a program

$$P ::= \texttt{while}(w > 0)\ \texttt{do}\ w := w - 1$$

and variable w is currently not speculated (i.e. takes on a definite value) and will not be updated by any other continuations in the system in future, then executing P at the a node higher in the tree has the same effect as executing multiple instances of P at the leaves of the sub-tree.

If a synchronous continuation σ attached to node $\mathcal{T}$ references variables that are currently materialized at $\mathcal{T}$, then operations can be done at node $\mathcal{T}$. E.g. if the operation is an assignment of a variable x, and $(x, v) \in \mathcal{DFV}(\mathcal{T})$, then $\mathcal{DFV}(\mathcal{T})$ is updated directly.

$$\mathcal{DFV}(\mathcal{T}), \sigma \xrightarrow{x:=v'} \mathcal{DFV}(\mathcal{T})[x = v'], \sigma'.$$

If the variable x referred to by σ, is not in $\mathcal{DFV}(\mathcal{T})$, then there are two cases. We discuss the actions to be taken in each case.

Firstly, if $(x, ?)$ does not imply $\mathcal{CP}(\mathcal{T})$, the "?" denotes any value, then the variable is materialized higher. One can locate x in the ancestor of $\mathcal{T}$, $\mathcal{T}'$, and redistribute x in the subtree so that node $\mathcal{T}$ now contains (x, v), and x is also copied to the highest possible descendent nodes of $\mathcal{T}'$ such that these nodes, together with $\mathcal{T}$, form a *frontier* around $\mathcal{T}'$. Then transition proceeds as usual.

Second, if $(x, ?)$ implies $\mathcal{CP}(\mathcal{T})$, it could either be in the ancestors of $\mathcal{T}$ or in the subtree of $\mathcal{T}$, as $\mathcal{CP}$ is only an approximation. We can locate it as in the first case if it is above $\mathcal{T}$. Otherwise, we have to clone the continuation σ because the x is under speculation and is taking on different values in the descendent nodes of $\mathcal{T}$. We will now attach one copy of σ at every node under $\mathcal{T}$ where x can be found in the $\mathcal{DFV}$ of that node, and remove σ from $\mathcal{T}$.

6.3 Reducing the size of multi-world

In this section, we introduce a strategy called *structure sharing* to contain the problem of compounding choices in the multi-world. The main idea of structure sharing is to keep the tree in linear form as much as possible and to maintain only pointers between the structures to indicate that a "cross product" relationship exists between them. We can linearize two subtrees, so long as the Σ's in one tree are data independent from the other. Data can to be moved from subtree to subtree to maintain the data independence.

We now demonstrate the strategy in figure 9 with the following example. Let us assume $\mathcal{DV}(\mathcal{T})$ contains four variables, $\{w, x, y, z\}$. Consider the following two programs.
$P1 ::= ((0 < z \leq 10) \Rightarrow z := z + 1) \oplus ((10 < z \leq 20) \Rightarrow z := z - 1)$
$P2 ::= (w > 0 \Rightarrow w := 1; z > 0 \Rightarrow z := 1) \oplus (w < 0 \Rightarrow w := 0)$

We represent the $\mathcal{DFV}$ by the names of the set of variables it contains in parentheses, and $\perp$ if the $\mathcal{DFV}$ is empty. P1$\rightarrow$ and P2$\rightarrow$ denote continuations from programs P1 and P2, and where they are attached to in the tree currently. Choice nodes are marked by unique ids. Diagrams (1) through (7) animate the steps of the multi-world transforming under structure sharing.

In the beginning of Fig. 9-(1) there is one store with four variables (w, x, y, z) and continuations from programs P1 and P2. In Fig. 9-(2), P1 starts first and makes a choice on z. P1 has to split to two copies as the continuation will be different in general. Since the first operation of P1 in both branches only requires z, only z is duplicated in two leaves and other variables are materialized in the $\mathcal{DFV}$ in the root. P2 executes next.

In Fig. 9-(3), P2 is issuing a new choice construct and its continuation is split in two. But instead of putting P2's choices on every leaf of node C_1, we construct a new subtree C_2 and put it side-by-side to C_1. Meanwhile, we create a pointer marked by "$\times$" from C_1 to C_2, to indicate that subtree C_2 is being "structure shared" by the leaf nodes of C_1. The semantics remains that C_2 is a *subordinate tree* on all leaves of C_1. And C_1 is said to be the *superior tree* of C_2. We write $\mathcal{T} \rightsquigarrow \mathcal{T}'$ to mean that $\mathcal{T}'$ is a subordinate of $\mathcal{T}$. If node C_1 already has a subordinate tree, we follow through the pointers to the terminal node, and create new pointer to C_2 from the terminal node. Thus any node in the tree has at most one subordinate tree.

In Fig. 9-(4), one branch of P2 requires variable w which is not with C_2. We trace back the pointer and look for w in the superior tree. In this case, w lives in node C_1 itself. We transfer w to C_2 and then the strategy in section 6.2 can be applied. In general, if w has n different values and lives in n nodes in C_1 (it lives in exactly n nodes because there is no redundancy in our tree), we will make a copy of C_2 for each node $\mathcal{T}_i$, such that $\mathcal{T}_i \rightsquigarrow C_2$, and transfer the values of w from $\mathcal{T}_i$ to the new C_2 trees.

Some time later (Fig. 9-(5)) in tree C_2, a guard is issued that requires z in the left branch. Because z is not in C_2, like in Fig. 9-(4), we search through C_1 till we realize that z is being speculated and has two values in two $\mathcal{DFV}$'s on the leaves. As we discussed above, since z has two values, we duplicate C_2 to two trees, and move z to the leaves of these tree for P2 to operate on. In this particular case, since C_{21} and C_{22} are the subordinate trees of the leaves of C_1, there is effective no more structure sharing, and the structure in Fig. 9-(5) is equivalent to the normal multi-world without optimization.

The next operation of P1 requires z but z is missing from the leaves. Since C_1 has no superior tree, we cannot bring z "down" to P1. Instead, we move it down to where the data is. Hence $P1$ is duplicated and moved to the leaves of C_{21} and C_{22}.

In Fig. 9-(6) and Fig. 9-(7), program P2 has committed in some leaves, deleted other branches and exit from the system. Eventually P1 has also committed in one of the branches and exit. The state of the system is reduced to just one definite store (w, x, y, z). Notice the sequence of (1) through (7) in Fig. 9 can happen any where in a multi-world, and not just the root.

To generalize, we delay the compounding of choice nodes by creating a single subordinate tree for the new choice, and structure share this subordinate tree. Logically the subordinate tree exists in the all the leaves of its superior tree. Data can move up this logical hierarchy by materializing differential views, and move down on demand by programs "living" underneath. Continuations are only duplicated and moved down the hierarchy if the data they require are not above them in the tree. The only exception is when all program continuations from the same program reaches and same program point, the data required are definite at the moment, and they are ready to execute the next step at the same time. Then these programs can be merged up and attached at their common root.

The following is a non-trivial but useful case when a multi-world can be linearized.

PROPERTY 1. *Given a set of programs R whose data requirements are* disjoint *from each other, the structure-shared GCC runtime structure is a linear ordering of $|R|$ subtrees, connected by "$\times$" pointers, where each subtree represents the runtime structure of a respective program.*

7. Simulation

To get a sense of how the implementation ideas fare in applications with speculation, we experimented with simulations based on the producer and consumer problem, and obtained some preliminary results. For simplicity, there are n_t types of resources being produced and consumed. We assume the producers' programs do not involve a choice, but just produce a number of resources of different types. The consumers' programs, on the other hand, are made up of just one GCC construct. In each branch, a series of consumptions of resources is done. Each consumption action tries to consume two items of the same type provided they are available, or else the consumer blocks until the items become available later. Each production action produces either 1, 2 or 3 items of the same type at a time.

In this simulation, computations are executed by clock ticks. At each clock tick, either a producer program is submitted to the system, or a consumer program is submitted, or nothing is submitted. At the same tick, every world in the system advances one step by picking one program (either producer or consumer) attached in this world and execute one of its next instruction.

We identify two dimensions of simulation characteristics:

- the level of overlapping interest by the producers and the consumers: high overlapping (HO) and low overlapping (LO)
- the relative rate of production against the consumption high production (HP), low(LP) and balanced production (BP)

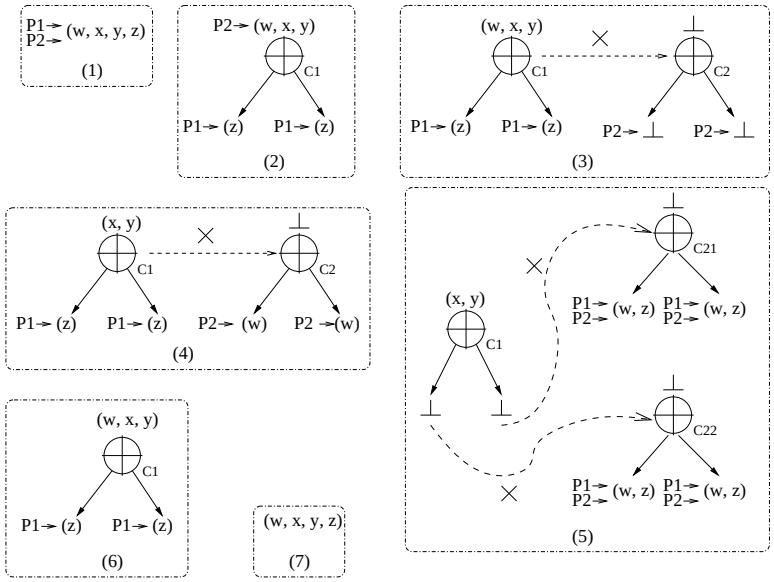


Figure 9. Example of structure sharing

	Max Nodes	Max Progs	Max Data Storage
HO/HP	24	17	6
HO/LP	66	45	6
HO/BP	15	10	7
LO/HP	57	39	51
LO/LP	111	75	63
LO/BP	105	70	22

Table 1. Simulation results

We thus create six datasets (HO/HP, HO/LP, HO/BP, LO/HP, LO/LP, and LO/BP) each consisting of 50 producers programs and 50 consumer programs are submitted to the system in random fashion. There is a random number (from 0 to 9) of clock ticks between two consecutive submissions. For HO, $n_t = 5$, while for LO, $n_t = 100$. Table 1 records the maximum number of tree nodes, maximum number of program continuations and the maximum size of the data storage (in terms of total number of (variable, value) pairs stored) in each experiment.

We see that although the interaction of 100 programs has the possibility of creating a very large multi-world, the optimization techniques in Section 6 are successful in controlling the multi-worlds and the storage requirements. The variation in the numbers in Table 1 have an easy and intuitive explanation due to the different nature of the datasets. Balanced production/consumption gives the smallest size tree, and smallest number of programs, while both excessive production and excessive consumption result in larger tree and more computation. The numbers for low-overlapping interest are larger than high-overlapping interest because the consumption and production are more likely to be mismatched given larger number of types of resources. These preliminary results demonstrate the "pay only when you use" principle behind our implementation techniques.

References

[1] A. J. Bonner and M. Kifer. Transaction logic programming. In *Proc. Intl. Conf. on Logic Programming*, 257–279, 1993.

[2] A. J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Joint Intl. Conf. and Symp. on Logic Programming*, 142–156, 1996.

[3] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *SIGMOD Conf.*, 202–214, 1990.

[4] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[5] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*, 249–259. ACM Press, 1987.

[6] G. Gupta, E. Pontelli, K. A. M. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel execution of prolog programs: a survey. *Trans. of Programming Languages and Systems*, 23(4):472–602, 2001.

[7] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Proc. of PPoPP*, 2005.

[8] C. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[9] M. E. C. Hull. Occam - a programming language for multiprocessor systems. *Comput. Lang.*, 12(1):27–37, 1987.

[10] J. Jaffar, R. H. Yap, and K. Q. Zhu. Coordination of many agents. In *Proc. of the Intl. Conf. on Logic Programming*, 98–112, 2005.

[11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *J. of Information and Computation*, 100:1–77, 1992.

[12] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, 2002.

[13] P. V. Roy, P. Brand, D. Duchier, S. Haridi, M. Henz, and C. Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. *TPLP*, 3(6):715–763, 2003.

[14] V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[15] E. Y. Shapiro and A. Takeuchi. Object oriented programming in concurrent prolog. *New Generation Comput.*, 1(1):25–48, 1983.

[16] G. Smolka. The Oz programming model. In *Computer Science Today*, 324–343. Springer, 1995.

[17] K. Ueda. Guarded horn clauses. In *Logic Programming*, LNCS 221, 168–179, 1986.

[18] I. Vlahavas and N. Bassiliades. *Parallel, Object-Oriented, and Active Knowledge-Base Systems.* Kluwer International Series on Advances in Database Systems. 1998.