УДК: 004.4'422

Название: Построение блоков обработки исключений при декомпиляции Java-байткода

Авторы:

Соловьев В.В. (Институт систем информатики им. А.П. Ершова СО РАН, Новосибирский государственный университет),

Липский Н.В. (Институт систем информатики им. А.П. Ершова СО РАН)

Аннотация: Задача декомпиляции Java-байткода состоит в построении исходного кода на языке Java, эквивалентного данному байткоду. Байткод — линейная программа с условными и безусловными переходами, а язык Java содержит структуры управления, который образуют иерархию в исходном коде. Эту иерархию необходимо восстанавливать при декомпиляции, в частности, необходимо восстановить блоки обработки исключений try-catch-finally. В проекте Excelsior RVM (виртуальной машины Java со статическим компилятором) байткод декомпилируется для проведения оптимизирующих преобразований. При построении блоков обработки исключений декомпилятор системы Excelsior RVM полагает, что байткод был получен путем компиляции исходного кода стандартным компилятором языка Java, и пытается совершить обратное преобразование. Иногда это не удается для байткода, полученного другими инструментами. В данной работе предложен алгоритм декомпиляции, восстанавливающий блоки обработки исключений из произвольного корректного байткода. Этот алгоритм реализован, интегрирован в систему Excelsior RVM и протестирован на реальных приложениях.

Ключевые слова: компиляция, декомпиляция, язык програмирования Java, Java-байткод, обработка исключений, блоки обработки исключений, таблицы защищенных интервалов

1. Введение. Исходный текст на языке Java [14] переводится Java-компилятором в двоичный формат (Java-байткод) виртуальной машины Java (JVM) [9]. Java-байткод является корректным, если он проходит процедуру верификации [9]. Рассматривается задача декомпиляции корректного байткода, а точнее — подзадача построения блоков обработки исключений. В декомпиляторе системы Excelsior RVM восстановление блоков обработки исключений предваряет восстановление прочих типов структур управления [1]. Таким образом, подзадачу построения блоков обработки исключений можно рассматривать независимо от прочих подзадач декомпиляции.

Блок обработки исключений языка Java выглядит следующим образом:

```
try { try-блок }
catch (CatchType1 e1) { catch-блок 1 }
```

```
…
catch (CatchType_N e_N) { catch-блок N }
finally { finally-блок }
```

Два блока обработки исключений не могут иметь общих catch-блоков и не могут текстуально пересекаться друг с другом кроме строгого вложения.

Таблица защищенных интервалов Java-байткода — это массив структур типа *интервал*. Каждый интервал — это четверка <StartPC, EndPC, HandlerPC, CatchType>, где StartPC, EndPC и HandlerPC — адреса инструкций в байткоде, а CatchType — идентификатор ссылочного типа. Для выражения отношения наследования будем применять символ <:, если X — наследник Y или равен Y, то будем записывать это как X <: Y. Семантика обработки исключений такова: если во время исполнения инструкции с адресом PC возникает исключение типа Type, то в таблице защищенных интервалов ищется первый интервал, удовлетворяющий свойствам: StartPC ≤ PC, PC < EndPC, CatchType <: Type. Если такой интервал находится, то управление переходит на инструкцию с адресом HandlerPC. Иначе исключение передается в метод, вызвавший данный. В корректном байткоде интервалы должны удовлетворять условиям: StartPC, EndPC, HandlerPC — корректные адреса инструкций; StartPC < EndPC; CatchType <: Throwable.

Стандартный компилятор языка Java транслирует блоки обработки исключений в таблицу защищенных интервалов по следующему алгоритму:

1) Try-блок компилируется в непрерывный участок байткода [SPC, EPC).
2) Catch-блок$_i$ компилируется в байткод, начинающийся с инструкции HPC$_i$.
3) Finally-блок компилируется по-разному в зависимости от версии компилятора
    a. Для версии Sun JDK 1.4 и более ранних finally-блок компилируется в байткод, который вызывается через инструкции jsr/ret из каждой точки выхода всех try, catch и finally блоков.
    b. В более поздних версиях компилятор копирует finally-блок и подставляет его в каждую точку выхода, выполняя, по сути, открытую подстановку подпрограммы, ранее реализуемую через jsr/ret.
4) В таблицу защищенных интервалов вставляются записи <SPC, EPC, HPC$_i$, CatchType$_i$> для каждого catch-блока. Также вставляются записи, покрывающие скомпилированный try-блок и все catch-блоки с HandlerPC, указывающим на finally-блок и CatchType, равным Throwable, для того чтобы finally-блок выполнялся в случае любого выхода.
5) Записи вносятся в таблицу в последовательности топологической сортировки по отношению текстуального вложения. Прежде чем будут внесены записи,

соответствующие некоторому блоку обработки исключений, в таблице уже должны быть записи, соответствующие всем блокам, вложенным в него.

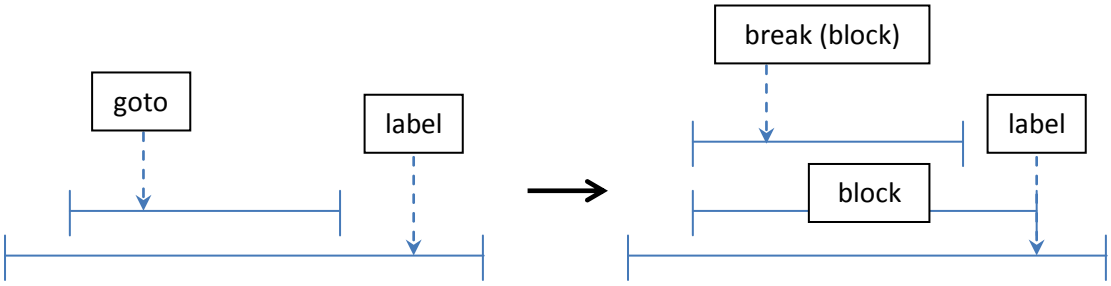
Предполагая, что таблица и байткод были получены стандартным компилятором языка Java, можно предложить *алгоритм прямой декомпиляции*:

1) try-блок определяется по StartPC и EndPC. Все записи таблицы, у которых совпадают эти границы, объединяются в одну группу;
2) для группы записей строится набор catch-блоков (все блоки, доминированные блоком, начинающимся с HandlerPC, попадают в catch-блок). Для этого строится Control Flow Graph [5];
3) в зависимости от того, как компилировался finally-блок, либо выбирается байткод, заключенный между jsr/ret инструкциями, либо ищется скопированный байткод, соответствующий finally-блоку;
4) выбранные группы байткода декомпилируются, и результат заключается в блок обработки исключений. CatchType_i берутся из таблицы интервалов;

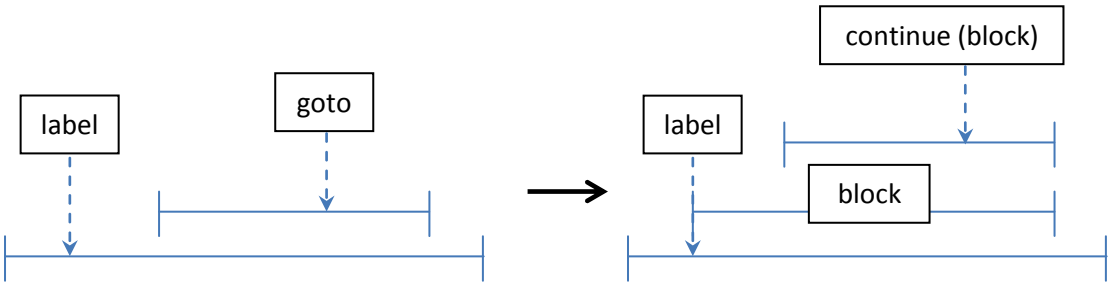
Произвольный байткод может быть получен с помощью обфускаторов, инструментации байткода, автоматической модификации или генерации кода (например, с помощью аспектно-ориентированного программирования), трансляцией JVM-based языков (JRuby, Jython, Scala) или программированием на Java ассемблере. Таблицы защищенных интервалов в таком байткоде могут не подходить для преобразования, обратного к преобразованиям, совершаемым стандартным компилятором языка Java. При постановке задачи декомпиляции произвольного корректного байткода можно полагаться только на ограничения, накладываемые верификатором. В частности, интервалы могут пересекаться без вложения одного в другого, у разных интервалов могут быть общие HandlerPC и т.п. Такие таблицы не могут быть декомпилированы прямым алгоритмом.

Результатом работы декомпилятора системы Excelsior RVM является дерево абстрактного синтаксиса (AST) [11], из которого можно получить исходный код на языке Java. В алгоритмах построения AST будет использоваться оператор goto. Чтобы представление оставалось структурным (возможность построения из него исходного кода), операторы goto должны моделироваться операторами break, continue и block языка Java. *Структурным оператором goto* будем называть оператор goto, который переводит исполнение на оператор label, если оператор label находится в произвольном месте цепочки, в которой находится или в которую вложен оператор goto (вложение может быть через несколько уровней). Структурные операторы goto можно смоделировать операторами break, continue и block. Это продемонстрировано на рис. 1. В части a) показано, как моделируется структурный оператор goto операторами break и block, в части

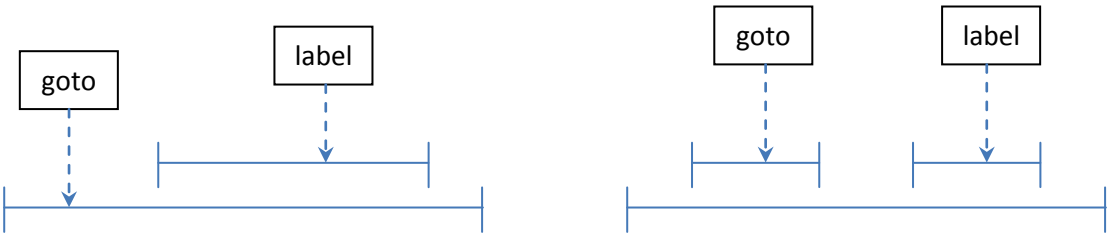
б) показано, как другой структурный оператор goto можно смоделировать операторами continue и block, в части в) показаны два примера неструктурного оператора goto.



а) Моделирование структурного оператора goto с помощью block и break



б) Моделирование структурного оператора goto с помощью block и continue



в) Примеры неструктурных переходов goto

Рис. 1

2. Формальная модель семантики обработки исключений. Разработанные нами алгоритмы декомпиляции блоков обработки исключений преобразовывают таблицы защищенных интервалов, поэтому нам необходимо доказать, что семантика обработки исключений сохраняется после преобразований. Для этого формализуем это понятие и введем ряд эквивалентных преобразований.

2.1. Определения. *Множество инструкций байткода* PC = 0, ..., L. PC $\subset$ N (натуральные числа). *Множество типов исключений* <XTypes, <:>, <: — частичный порядок, означающий наследование. Наименьший элемент XTypes — Throwable. *Множество интервалов* XIntervals = {<S, E, T, H>}, S, E, H $\in$ PC, S < E, T $\in$ XTypes.

Говоря об интервале A, равном <S, E, T, H>, будем обозначать S, E, T и H, как A.Start, A.End, A.Type и A.Handler соответственно. Целочисленный интервал [A.Start, A.End) будем обозначать A.Range. *Множество таблиц исключений* XTables = {{Int_i | Int_i ∈ XIntervals, i = 0..k}} — множество всех конечных упорядоченных последовательностей элементов из XIntervals. Говоря о таблице X ∈ XTables, X = (Int_1, …, Int_k), будем обозначать Int_i, как X_i, а k как |X|. *Предикат риска* RF: PC → {0, 1}. RF(X) = 1 ⇔ инструкция с адресом X может вызывать исключение. *Безопасный код* NRF = {c ∈ PC, RF(c) = 0}. *Предикат безопасности* UTF: XIntervals → {0, 1}. UTF(X) = 1 ⇔ X.Range ⊂ NRF. *Функция интерпретации*: Δ: XTables x (PC / NRF) x XTypes → PC ∪ {⊥}

Δ(<XT, C, Type>) =

1)XT_i.Handler, где i = min{i | C ∈ XT_i.Range и Type <: XT_i.Type};

2)⊥, если такого i нет.

Отношение эквивалентности на XTables: XT_1 ≈ XT_2 ⇔ ∀ C ∈ PC/NRF, ∀ type ∈ XTypes: Δ(XT_1,C,type) = Δ(XT_2,C,type). Преобразование Ψ: XTables → XTables будем называть *эквивалентным преобразованием*, если ∀ XT ∈ XTables: XT ≈ Ψ (XT).

2.2. Система эквивалентных преобразований. Определим три преобразования. Первое — Sep, разбивает некоторый интервал таблицы по некоторой инструкции и вставляет два подинтервала в таблицу вместо исходного. Второе — Del, удаляет интервал из таблицы, если байткод, ограниченный этим интервалом, не может бросать исключения. Третье — Concat, объединяет два подряд идущих интервала в таблице, если между ними находится код, который не может бросать исключения. Интуитивно понятно, что эти преобразования сохраняют семантику обработки исключений, поэтому формальное доказательство этого факта будет приведено в приложении A, а в дальнейшей работе будем полагаться на его справедливость.

Пусть XT ∈ XTables. Преобразование $Sep_{n,c}$(XT), где c ∈ (XT_n.Start, XT_n.End) – *разбиение n-ого интервала таблицы по инструкции c*, по следующему правилу:

$Sep_{n,c}$ (XT) = SepT такая, что $SepT_i$ =

1)XT_i, если i < n

2)<XT_n.Start, C, XT_n.Type, XT_n.Handler>, если i = n

3)< C, XT_n.End, XT_n.Type, XT_n.Handler>, если i = n + 1

4)XT_{i+1}, если i > n + 1

Преобразование $Del_n(XT)$, где $UTF(XT_n) = 1$ — *удаление безопасного интервала n*, по следующему правилу:

$Del_n(XT)$ = DelT такая, что $DelT_i$ =

1)XT_i, если $i < n$

2)XT_{i+1}, иначе

Преобразование $Concat_n(XT)$, где $XT_n.End < XT_{n+1}.Start$, $XT_n.Type = XT_{n+1}.Type$, $XT_n.Handler = XT_{n+1}.Handler$, $[XT_n.End, XT_{n+1}.Start) \subset NRF$ – *слияние n-ого и n+1-ого безопасно разделенных интервалов*, по следующему правилу:

$Concat_n(XT)$ = ConcatT такая, что $ConcatT_i$ =

1)XT_i, если $i < n$

2)$<XT_n.Start, XT_{n+1}.End, XT_n.Type, XT_n.Handler>$, если $i = n$

3)XT_{i+1}, если $i > n$

Утверждение 1: Преобразования $Sep_{n,c}$, Del_n, $Concat_n$ являются эквивалентными.

3. Алгоритм декомпиляции произвольной таблицы защищенных интервалов. Алгоритм прямой декомпиляции хорошо работает с таблицами защищенных интервалов, полученными компиляцией исходного текста Java-компилятором. В основе его работы лежит предположение, что таблица была получена прямой компиляцией, поэтому он накладывает на нее множество условий, которые обеспечивают полное восстановление блоков обработки исключений. В произвольной корректной таблице эти условия могут не выполняться, и поэтому возникают проблемы прямой декомпиляции.

3.1. Декомпиляция try-блоков. В произвольной корректной таблице два интервала могут пересекаться без вложения одного в другой (Рис. 2а). Такая таблица не может быть получена в результате прямой компиляции исходного текста на языке Java и, соответственно, не может быть напрямую декомпилирована. Также, если в исходном тексте один блок обработки исключений был вложен в другой, то при компиляции в байткод, интервалы вложенного блока будут находиться в таблице перед интервалами объемлющего. Таким образом, при выбросе исключения виртуальная машина сначала проверит интервалы вложенного блока, затем объемлющего. В то же время в верификаторе нет такого требования, и объемлющий интервал может находиться в таблице раньше, чем вложенный (Рис. 2б). Если декомпилировать оба таких интервала в блоки обработки исключений, то их вложенность будет нарушать семантику обработки исключений в исходном байткоде.

Далее будем обозначать таблицу исключений текущего декомпилируемого метода как XT. XT ∈ XTables. XT_i *пересекается без вложения* с XT_j, если XT_i и XT_j пересекаются, но ни один не вкладывается в другой. XT_i *неправильно вкладывается* в XT_j, если XT_i строго вкладывается в XT_j, и при этом i > j. Таблицу исключений будем называть *регулярной*, если в ней нет пересечений без вложения и неправильных вложений.

а) Пересечение без вложения б) Неправильное вложение

Рис. 2

3.1.1. Алгоритм проверки таблицы на регулярность. Алгоритм, проверяющий для произвольной таблицы, является ли она регулярной, на псевдоязыке высокого уровня выглядит следующим образом:

```
Stack<ExcepInfo> stack;      -- стек вложенности
Sort(XT);                    -- сортируем таблицу (по увеличению Start и, при
                             -- равных Start, по уменьшению End)
stack.push(XT[0]);           -- кладем на стек первый интервал
FOR i := 1 TO Len(XT)-1 DO
    WHILE ((stack.size > 0) AND (XT[i]  не пересекается со stack.top)) DO
        stack.pop;        -- удаляем, пока не встретим пересечение интервалов
    END;
    IF ( (stack.size > 0) AND
           ( (XT[i] пересекается без вложения со stack.top) OR
             (XT[i] неправильно вложен в stack.top))) THEN
        RETURN FALSE;         -- ситуация, когда stack.top строго вложен в
                              -- XT[i] исключена способом сортировки
    END;
    stack.push(XT[i]);        -- новая верхушка стека
END;
RETURN TRUE;
```

Утверждение 2: Алгоритм проверки таблицы на регулярность определяет, является ли таблица XT регулярной и обладает ли следующими характеристиками: трудоемкость — O(|XT| * ln(|XT|)), емкостная сложность — O(|XT|). Доказательство в приложении А.

3.1.2. Алгоритм приведения таблицы к регулярному виду. Мы собрали статистику на тестовом наборе системы Excelsior RVM и изучили примеры байткода, таблицы исключений в которых были нерегулярными. Обнаружилось, что такие таблицы

появляются в основном в обфусцированных программах либо в программах, где finally-блок был скомпилирован с применением jsr/ret инструкций. Когда finally-блок компилируется с применением jsr/ret инструкций, то инструкция jsr, вставленная внутрь try-блока, с точки зрения компилятора Java не должна включаться в защищенный интервал. Поэтому оригинальный try-блок разрезается этими инструкциями на несколько записей в таблице исключений. При этом может появиться неправильная вложенность. Еще одной частой причиной нерегулярности являются интервалы, такие что HandlerPC = StartPC и область кода, которую они защищают, неспособна создать исключение. Их удаление из таблицы исключений не влияет на семантику обработки.

Алгоритм приведения таблицы к регулярному виду

1) Применим преобразование $Del_i(XT)$ для всех i таких, что UTF(XT[i]) = 1 и XT[i].Handler = XT[i].Start;
2) Применим преобразование $Concat_i(XT)$ для всех подходящих i.

Утверждение 3: Алгоритм приведения таблицы к регулярному виду сохраняет семантику обработки исключений, что следует из утверждения 1. Его временная сложность составляет O(|XT|), так как оба его шага осуществляются линейным проходом по таблице исключений, а каждое преобразование выполняется за O(1). Емкостная сложность алгоритма составляет O(1).

3.1.3. Алгоритм уравнивания интервалов. После реализации методов преобразования таблиц, мы повторно собрали статистику и обнаружили, что во всем тестовом наборе остался только один метод, таблица исключений в котором остается нерегулярной. Кроме поставленной практической задачи, которую можно считать разрешенной, была поставлена и теоретическая задача о декомпиляции произвольной таблицы исключений. Поэтому для покрытия всех случаев, нами был разработан алгоритм преобразования, работающий в тех случаях, когда таблица не является регулярной и не приводится к регулярному виду предыдущим алгоритмом.

Алгоритм уравнивания интервалов

1. Все пересечения разбиваются преобразованиями $Sep_{i,c}(XT)$ где C – границы пересекающихся интервалов.
2. Применим преобразование $Del_i(XT)$ для всех i таких, что UTF(XT[i]) = 1, так как в результате первого пункта могут возникнуть такие интервалы.

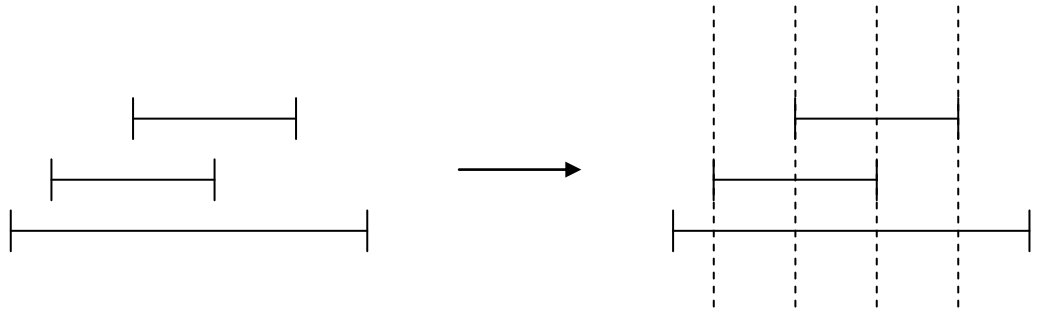


Рис. 3

Утверждение 4: Алгоритм уравнивания интервалов сохраняет семантику обработки исключений, что следует из утверждения 1. Таблица исключений после такого преобразования станет регулярной, так как все интервалы в преобразованной таблице будут либо совпадать, либо не пересекаться вообще. Каждый интервал можно порезать максимум 2*(|XT|-1) точками, причем множество этих точек можно построить заранее. Используя линейный список для хранения получаемой в процессе алгоритма таблицы, сделаем максимум |XT| *2*(|XT| -1) элементарных разбиений и |XT| *(2*|XT| + 1) операций вставки в линейный список. Таким образом, временная сложность O(|XT| *|XT|). Емкостная сложность также равна O(|XT| *|XT|).

3.2. Декомпиляция catch-блоков. В блоках обработки исключений управление в catch-блок может перейти только из соответствующего ему try-блока. В тоже время в таблице защищенных интервалов два разных интервала могут указывать на один и тот же HandlerPC, что делает невозможным прямую декомпиляцию такого байткода. Также HandlerPC может указывать в середину другого try-блока или другого catch-блока, что также невозможно представить в исходном тексте и в структурном представлении программы.

Основной идеей описываемого алгоритма декомпиляции является отказ от декомпиляции catch-блоков в смысле полного восстановления их в блок обработки исключений как цепочек операторов. Вместо этого во внутреннем представлении создается следующая структура:

```
try { try-блок }
catch (CatchType1 e1) { goto handler1; }
…
catch (CatchTypeN eN) { goto handlerN; }
```

$Handler_i$ — метка, соответствующая началу декомпилированных операторов, начинающихся с HandlerPC i-ой записи таблицы. При такой декомпиляции исчезают проблемы с совпадающими catch-блоками, принадлежащими разным интервалам. При этом построении нам нужно проверять, что операторы goto Handler являются

структурными операторами goto. XT[i].Handler является *структурным*, если для любого XT[j], такого что XT[i].Handler лежит в интервале (XT[j].Start, XT[j].End), XT[i] строго вложен в XT[j]. Нам необходимо, чтобы все Handler в таблице исключений были структурными. Проверка этого условия осуществляется за O(|XT|*|XT|).

Алгоритм разбиения интервалов

1. Проходим по таблице и проверяем для каждой пары интервалов X и Y, где X.Handler попадает внутрь Y, является ли X строго вложенным в Y. Если не является, то применяем преобразование $Sep_{i,X.Handler}(XT)$, где i – номер Y.
2. Применяем преобразование $Del_i(XT)$ для всех i таких, что UTF(XT[i]) = 1, так как в результате первого пункта могут возникнуть такие интервалы.

Утверждение 5: Временная и емкостная сложности данного алгоритма такие же, как и у алгоритма уравнивания интервалов — O(|XT| * |XT|). Алгоритм разбиения интервалов сохраняет семантику обработки исключений, что следует из утверждения 1. Этот алгоритм сохраняет регулярность таблицы, так как разбиение каждой новой границей (HandlerPC интервала X) разбивает все интервалы (кроме тех, в которые X строго вложен), и пересечений и неправильных вложений не появляется (рис. 4). Этот алгоритм увеличивает таблицу. В худшем случае (рис. 4), таблица увеличивается в 2*|XT| раз. Однако даже в таком теоретическом случае количество блоков обработки исключений останется 2*|XT|.

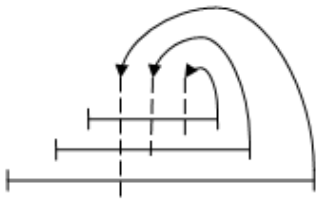


Рис. 4

3.3. Декомпиляция finally-блоков. Декомпиляция finally-блоков в случае их реализации в байткоде через инструкции jsr/ret затруднена тем, что блоки обработки исключений могут быть вложены друг в друга. Семантика обработки исключений предполагает в таком случае последовательное исполнение finally-блоков от различных блоков в порядке их вложенности друг в друга, что может быть неверно в произвольном байткоде. Также в результате оптимизирующих или обфусцирующих преобразований в некоторых точках выхода из try-блока может не выполняться finally-блок. При открытой подстановке finally обратное преобразование затруднено тем, что различные копии finally-блока, подставленные в точки выхода из try и catch блоков, могут претерпеть оптимизирующие или обфусцирующие изменения под воздействием инструмента,

порождающего байткод. При декомпиляции требуется проводить дополнительный анализ и доказывать, что предполагаемые участки кода можно декомпилировать в один finally-блок, что в общем случае алгоритмически неразрешимо.

Рассматривается два случая: открытая подстановка finally и использование инструкций jsr/ret. В первом случае try-блок исходного текста, разделенный в байткоде копиями finally-блока, представлен в таблице исключений несколькими записями (рис. 5). Этот набор записей имеет одинаковые HandlerPC, то есть одинаковые обработчики исключений, и, если не объединить этот набор записей в оригинальную единственную структуру, то невозможно декомпилировать таблицу исключений методом прямой декомпиляции. Однако в разделе 3.2 уже решена проблема декомпиляции таблиц исключений с совпадающими HandlerPC, так что необязательно проводить декомпиляцию finally. Как видно на рис. 6, полученные копии finally-блока можно рассматривать как участки кода, не имеющие никакого отношения к декомпилируемому блоку обработки исключений, и декомпилировать их отдельно.

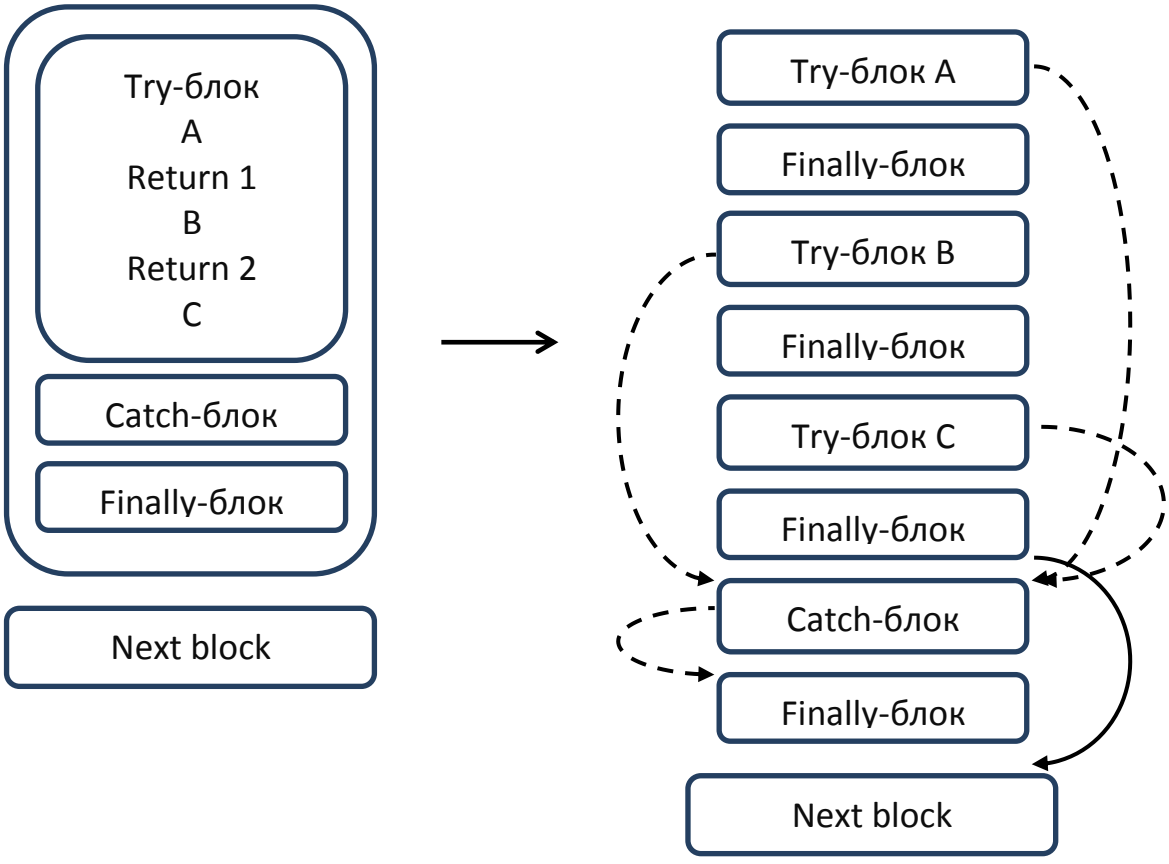


Рис. 5. Результат преобразования блока обработки исключений в результате открытой подстановки finally. Пунктирные дуги соответствуют записям в таблице защищенных интервалов, сплошная дуга — безусловный переход.

Во втором случае, когда finally-блок компилируется с использованием jsr/ret инструкций, для каждой инструкции ret определяется множество соответствующих ей инструкций jsr. Для каждой такой группы, представляющей один finally-блок, заводится локальная переменная local_X. Каждая инструкция jsr_N представляется таким образом:

```
local_X := N;
goto jsr_target;
```

Инструкция ret представляется в AST структурой switch-goto:

```
switch (local_X) {
case 1: goto jsr_1.next;
…
case N: goto jsr_N.next; }
```

Здесь jsr_N.next – оператор, следующий за инструкцией jsr_N. При этом построении нужно так же, как и в алгоритме разрешения проблемы декомпиляции catch-блоков, следить за структурностью создаваемых операторов goto.

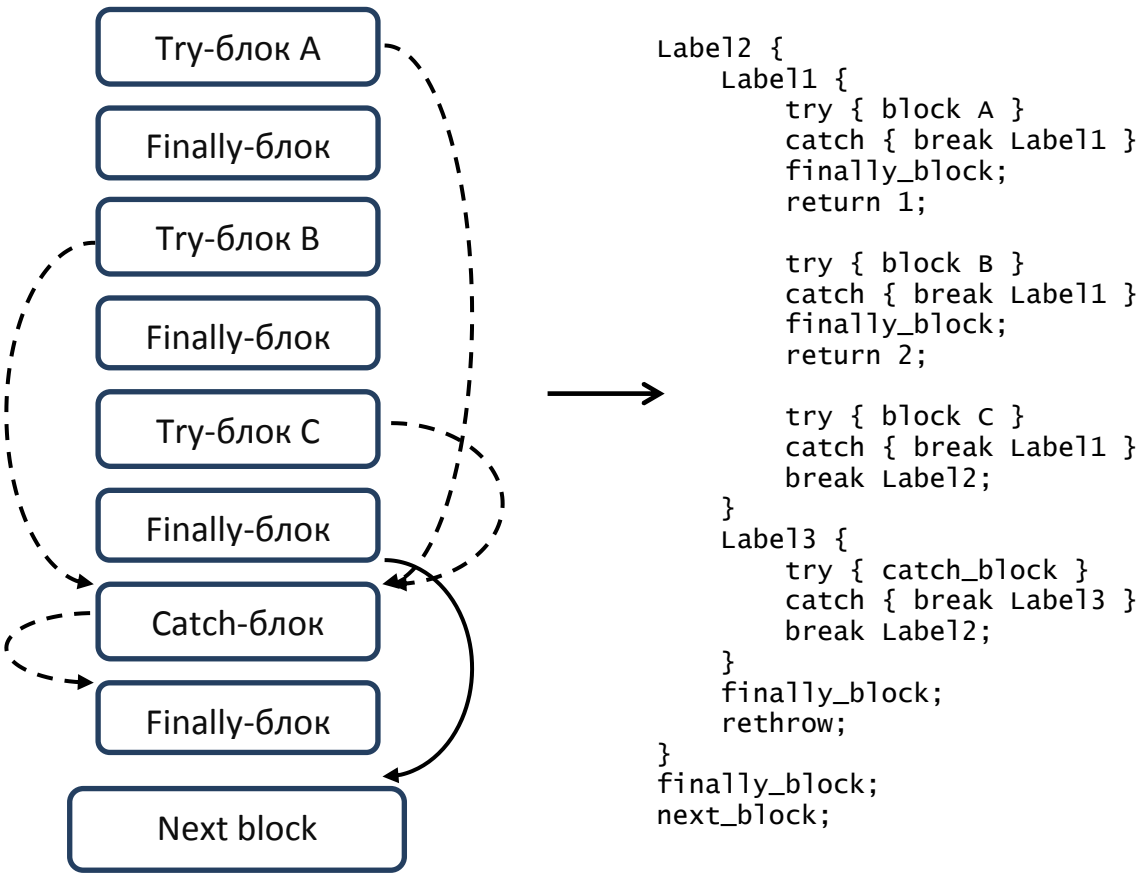


Рис. 6

Алгоритм декомпиляции jsr/ret

1. Выделяется группа операций jsr, соответствующих одной операции ret.
2. Для каждой такой группы заводится дополнительная локальная переменная.

3. Инструкции jsr декомпилируются в оператор, заполняющий значение локальной переменной уникальным значением, и переход на аргумент jsr.
4. Инструкции ret декомпилируются в конструкции switch, каждый case которых по значению локальной переменной осуществляет переход на инструкцию, следующую за инструкцией jsr, соответствующей этому значению.

Утверждение 6: Алгоритм корректен по построению и семантике исполнения jsr/ret инструкций. Трудоемкость алгоритма — O(code), где code — размер декомпилируемого байткода. Это так, потому что первый шаг алгоритма проводится верификатором, который работает до декомпилятора, и можно использовать результаты его работы, а количество всех остальных шагов не может превышать количества инструкций jsr в коде. Емкостная сложность также составляет O(code).

3.4. Общий алгоритм декомпиляции блоков обработки исключений. Общий алгоритм использует все вышеописанный алгоритмы и генерирует структурное представление обработки исключения для произвольной таблицы исключений. Структура AST, соответствующая одному блоку обработки исключений, содержит try-блок и список обработчиков catches. Назовем эту структуру TCB (Try-Catch-Block). Исполнение TCB — это исполнение его блока try. Если во время этого исполнения возникает исключение типа Type, то в списке catches ищется первый обработчик, подходящий по типу исключения. Если он находится, то исключение считается обработанным, и исполняется оператор goto Handler_label. Иначе исключение считается необработанным и передается дальше в структуру, в которую вложена структура TCB.

Общий алгоритм декомпиляции

1. Если таблица нерегулярная, применяем алгоритм приведения таблицы к регулярному виду и, при необходимости, алгоритм уравнивания интервалов.
2. Если есть неструктурные Handler, применяем алгоритм разбиения интервалов.
3. Если в коде есть инструкции jsr/ret, применяем алгоритм декомпиляции jsr/ret.
4. Выделяем в таблице группы интервалов, совпадающих по Start и End. Создаем структуру TCB, try-блок которой получается декомпиляцией интервала [Start, End), а список catches создается из выделенной группы записей в том порядке, в котором они находятся в таблице.
5. Для каждого catch из списка catches порождаем оператор goto, аргумент которого — метка на операторе Handler соответствующего catch.

Трудоемкость и емкостная сложность алгоритма: Используя оценки, полученные ранее, можно сказать, что трудоемкость общего алгоритма декомпиляции равна O(|XT|*|XT| + code). Теоретическая сложность действительно достигает указанной оценки,

но на практике, во-первых, |XT| << code, во-вторых, алгоритм уравнивания интервалов не применяется, в-третьих, алгоритм разбиения интервалов не увеличивает размер таблицы квадратично. К тому же в задаче декомпиляции было бы разумно оценивать трудоемкость в зависимости от размера кода. Таким образом, практической средней оценкой трудоемкости, подтвержденной статистическими данными, будем считать O(code). Емкостная сложность — O(|XT| * |XT| + code), на практике — O(code).

Утверждение о корректности: Семантика обработки исключений декомпилируемой таблицы совпадает с семантикой обработки исключений в структуре TCB, полученной общим алгоритмом декомпиляции. Доказательство приведено в приложении A.

Декларация об универсальности: Общий алгоритм декомпиляции применим для произвольной корректной таблицы защищенных интервалов, т.к. не накладывает на нее никаких ограничений, кроме тех, которые наложены верификатором.

4. Реализация и эксперименты. Общий алгоритм декомпиляции блоков обработки исключений встроен в алгоритм декомпиляции Java-байткода системы Excelsior RVM как дополнительный для уже существующего. Это означает, что он применяется для декомпиляции тех методов, в которых не удается декомпилировать таблицы исключений алгоритмом прямой декомпиляции. Если во время попытки провести прямую декомпиляцию происходит ошибка, то алгоритм возвращается к оригинальным байткоду и таблице исключений и применяет разработанный нами алгоритм. Решение о такой интеграции основано на том, что если есть возможность полного восстановления блоков обработки исключений, то лучше реализовать ее, так как у метода прямой декомпиляции меньше издержки на размер генерируемого внутреннего представления.

Мы сравнили результаты компиляции тестового набора приложений в схеме, в которой работал только алгоритм прямой декомпиляции (эталонный режим), со схемой, когда разработанный нами общий алгоритм декомпиляции был дополнительным для прямого алгоритма (тестируемый режим). Экспериментальные результаты были сведены в таблицу 1, которая содержит следующие колонки: название приложения (приложение), общее количество методов (кол-во методов), количество методов с непустыми таблицами исключений (кол-во таблиц), количество методов, не декомпилированных эталонным режимом (отказы эталона по TCF), количество методов, не декомпилированных обоими режимами по причинам, не связанным с декомпиляцией таблиц исключений (отказы другого рода). В тестируемом режиме для всех приложений были декомпилированы все методы, кроме тех, которые попали в группу «отказов другого рода». В эту группу попадают методы, размер которых полагается компилятором слишком большим для проведения оптимизаций. В тестовый набор были включены крупные Java-приложения и

библиотеки. Среди них можно выделить: Eclipse RCP приложения, систему учета ошибок JIRA, экспериментальную реализацию Java SE — Harmony, реализацию языка Ruby поверх JVM — JRuby, тесты совместимости со спецификацией Java SE — JCK6.

Приложение	**Кол-во методов**	**Кол-во таблиц**	**Отказы эталона по TCF**	**Отказы другого рода**
Eclipse-3.4-jee	381088	25763	1228	7
Eclipse Europa	237196	18669	854	0
Apache Directory Studio	124359	8051	259	0
EAStudio 1.5.1	248045	15784	333	0
Eclipse-3.2	193108	15890	21	0
Escape-K	134765	10128	261	0
MyEclipse_6.0	280787	20442	713	2
MyTourbook 1.5.0	125988	10048	195	0
Relations	105750	8342	8	0
RSSOwl 2.0	61835	4195	144	0
SafiWorkshop 1.0.4	309080	23175	354	1
XMIND 2.3	140961	8274	224	0
Harmony6.0	105223	8562	55	2
Jira 3.12.1	130492	11751	355	0
JRuby-1.1RC1	28256	4381	1998	0
JCK6b	323691	21835	107	0

Таблица 1

На основе этих результатов можно сказать, что невозможность декомпиляции таблицы исключений была основной причиной невозможности декомпиляции байткода в целом. Это подтверждает экспериментально то, что с помощью общего алгоритма декомпиляции таблиц исключений можно построить для произвольного верифицируемого байткода эквивалентное ему структурное представление, т.е. эквивалентную программу на языке Java. Таким образом, на практике подтверждена равномощность Java-байткода и языка Java. Также положительным результатом оказалось то, что для приложения с наибольшим отношением кол-ва методов к количеству отказов эталона — JRuby-1.1RC1, применение тестируемого режима дало прирост производительности порядка 15% за счет лучших оптимизирующих преобразований декомпилированного байткода. Также среди

результатов можно отметить, что система Excelsior RVM в тестируемом режиме прошла Sun JCK 6b (Java Compatibility Kit).

5. Заключение. Началом работ по декомпиляции можно считать работы Дейкстры [6], [7], [13], разработавшего основные положения структурного программирования и развивавшего идею отказа от использования оператора goto. Ему принадлежит формулировка и доказательство теоремы: «Алгоритм любой сложности можно реализовать, используя только три конструкции: следования (линейные), выбора (ветвления) и повторения (циклические).» [6]. Следствием этой теоремы является то, что любой байткод (программу с операторами goto) можно преобразовать в высокоуровневое представление без операторов goto (в нашем случае — только со структурными операторами goto).

Декомпиляцией исполняемого кода процессора i80286 в высокоуровневое представление и затем в исходный текст на языке C занималась Кристина Сайфуентис. Она описала строение и этапы работы абстрактного декомпилятора [2], [3], анализ информационных зависимостей и оптимизацию для указанного процессора [4], а также предложила алгоритмы декомпиляции циклов и условных операторов на основе построения управляющего графа (CFG) [5]. Ее работа была посвящена декомпиляции в исходный текст языка C, поэтому декомпиляция блоков обработки исключений, которых в этом языке нет, не рассматривалась.

Управляющий граф в том виде, в котором он представлен в работах Сайфуентис, не подходит для декомпиляции во внутреннее представление, включающее структуры обработки исключений, т.к. добавление дуг, по которым переходит управление в случае возникновения исключения, нарушает анализ потока управления. В работе Джанг-Ву Йо и Бьенг-Мо Чанг было предложено разбиение управляющего графа, включающего дуги обработки исключений, на Normal Control Flow Graph (NCFG) и Exception-Induced Control Flow Graph (EICFG) [8]. Ими была показана корректность такого разбиения и возможность отдельного анализа NCFG и EICFG.

Авторитетными специалистами в области декомпиляции Java-байткода являются сотрудники канадского университета McGill Джером Мицниковски и Лори Андран. В их статьях рассмотрен алгоритм декомпиляции с использованием внутренних представлений CFG и SET (дерева структурного вложения) [10]. Что касается конкретных проблем с декомпиляцией блоков обработки исключений, то ими также замечено то, что верифицируемые таблицы исключений Java-байткода могут быть неструктурными, что не покрывается предлагаемым ими алгоритмом декомпиляции, однако ими не было предложено какого-либо полного разрешения этих проблем.

Декомпилятор Java-байткода системы Excelsior RVM описан в работе Липского [1]. Он включает в себя алгоритм декомпиляции блоков обработки исключений, основанный на методе прямой декомпиляции с восстановлением полной структуры обработки исключений. Мотивацией для настоящей работы стало то, что с момента реализации исходного алгоритма появилось множество инструментов создания Java-байткода, создающих его таким, что он не удовлетворяет условиям структурности. Из-за этого многие популярные приложения, существующие в виде Java-байткода, стало невозможно декомпилировать в высокоуровневое внутреннее представление.

Данная работа дополняет все предыдущие тем, что в ней классифицированы проблемы декомпиляции блоков обработки исключений, предложены алгоритмы их разрешения, и проведены статистические исследования. Некоторой особенностью подхода является то, что целью было именно построение структурного представления, необходимого для проведения оптимизирующих преобразований, а не исходного текста, как в статьях Сайфуентис и Мицниковски. Такая постановка задачи не требовала генерации структурного представления, позволяющего получить читаемый исходный текст, из которого был получен декомпилируемый байткод. Это позволило беспрепятственно использовать преобразования таблиц исключений. Несмотря на это, из структурного представления, генерируемого описанным алгоритмом декомпиляции, также можно построить и исходный текст на языке Java.

Данная работа вместе с предшествующей работой Липского [1] подтверждает равномощность Java-байткода и языка Java — для произвольного верифицируемого байткода можно построить семантически эквивалентную ему программу на языке Java. Таким образом, задача декомпиляции Java-байткода полностью решена.

Список литературы

1. Липский Н.В. Квалификационная работа магистра «Декомпилятор байт-кода виртуальной Ява машины». НГУ, 1998.
2. Cifuentes, C. A Methodology for Decompilation / C. Cifuentes, K. J. Gough // Proceedings of the XIX Conferencia Latinoamericana deInformatica. – Buenos Aires, Argentina: 1993. – P. 257-266.
3. Cifuentes, C. A structuring algorithm for decompilation / C. Cifuentes // In XIX Conferencia Latinoamericana de Inform'atica. – 1993. – P. 267-276.
4. Cifuentes, C. Interprocedural data flow decompilation / C. Cifuentes // Journal of Programming Languages. – 1996. – Vol. 4. – P. 77-99.
5. Cifuentes, C. Structuring decompiled graphs / C. Cifuentes // In Proceedings of the International Conference on Compiler Construction. – Springer Verlag, 1996. – P. 91-105.

6. Dijkstra, E. W. A Discipline of Programming / E. W. Dijkstra. – 1st edition. – Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.
7. Dijkstra, E. W. Classics in software engineering / E. W. Dijkstra / Ed. by E. N. Yourdon. – Upper Saddle River, NJ, USA: Yourdon Press, 1979. – P. 1-9.
8. Jo, J.-W. Constructing control flow graph that accounts for exception induced control flows for java / J.-W. Jo, B.-M. Chang // Science and Technology, 2003. Proceedings KORUS 2003. The 7th Korea-Russia International Symposium on. – Vol. 2. – 2003. – P. 160-165.
9. Lindholm, T. Java Virtual Machine Specification / T. Lindholm, F. Yellin. – 2nd edition. – Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
10. Miecznikowski, J. Decompiling Java using staged encapsulation / J. Miecznikowski, L. Hendren // Reverse Engineering, 2001. Proceedings. Eighth Working Conference on. – 2001. – P. 368-374.
11. Muchnick, S. S. Advanced compiler design and implementation / S. S. Muchnick. – San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
12. Overview of Excelsior Jet, a high performance alternative to Java virtual machines / V. Mikheev, N. Lipsky, D. Gurchenkov et al. // Proceedings of the 3rd international workshop on Software and performance. – WOSP '02. – New York, NY, USA: ACM, 2002. – P. 104-113.
13. Structured programming / Ed. by O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare. – London, UK, UK: Academic Press Ltd., 1972.
14. The Java Language Specification, Third Edition / J. Gosling, B. Joy, G. Steele, G. Bracha. – Prentice Hall, 2005.

Приложение А.

Доказательство утверждения 1

1. Пусть $XT \in XTables$, $ST = Sep_{n,c}(XT)$, $P \in PC/NRF$, $T \in XTypes$. Пусть $\Delta(XT, P, T) = X$. Покажем, что $\Delta(ST, P, T) = X$.

1.1 Если $X \in PC$, то $\exists\, i$: $P \in XT_i.Range$, $T <: XT_i.Type$, причем это i — минимальное, $X = XT_i.Handler$. Если $i < n$, то $\Delta(ST, P, T) = X$, так как $ST_i = XT_i$. Пусть $i = n$. Тогда $P \in XT_n.Range$. Если $P < C$, то $P \in [XT_n.Start, C) = ST_n.Range$. Если $P \geq C$, то $P \in [C, XT_n.End) = ST_{n+1}.Range$. В любом случае, $\Delta(ST, P, T) = ST_n.Handler = XT_n.Handler = X$. Если $i > n$, то $\Delta(ST, P, T) = X$, так как $ST_{i+1} = XT_i$.

1.2 Если $X = \bot$, то $\nexists\, i$: $P \in XT_i.Range$ и $T <: XT_i.Type$. Предположим: $\Delta(ST, P, T) \in PC$. Тогда $\exists\, i$: $P \in ST_i.Range$, $T <: ST_i.Type$. Если $i < n$ или $i > n+1$, то $ST_i = XT_i \Rightarrow \Delta(XT, P, T) = \Delta(ST, P, T) \in PC$ — противоречие. Если $i = n$ или $i = n+1$, то $P \in ST_n.Range \cup ST_{n+1}.Range =$

XT_n.Range. Так как T <: ST_n.Type = ST_{n+1}.Type = XT_n.Type, то Δ(XT, P, T) = Δ(ST, P, T) ∈ PC — противоречие. Δ(ST, P, T) ∉ PC ⇒ Δ(ST, P, T) = ⊥.

Из произвольности P и T следует, что ST ≈ XT.

2. Пусть XT ∈ XTables, UTF(XT_n) = 1, DT = Del_n(XT), P ∈ PC/NRF, T ∈ XTypes. Пусть Δ(XT, P, T) = X. Покажем, что Δ(DT, P, T) = X.

2.1 Если X ∈ PC, то ∃ i: P ∈ XT_i.Range, T <: XT_i.Type, X = XT_i.Handler. (UTF(XT_n) = 1) ⇒ (∀ C ∈ XT_n.Range: RF(C) = 0) ⇒ (XT_n.Range ⊂ NRF) ⇒ (P ∉ XT_n.Range) ⇒ (i ≠ n). Если i < n, то DT_i = XT_i. Если i > n, то DT_{i-1} = XT_i. В любом случае, Δ(DT, P, T) = X.

2.2 Если X = ⊥. Предположим: Δ(DT, P, T) ∈ PC. Тогда ∃ k: P ∈ DT_k.Range. ∀ k = 0,..., |DT|: ∃ i такое, что DT_k = XT_i. (T <: DT_k.Type) ⇒ (∃ i: P ∈ XT_i.Range, T <: XT_i.Type) ⇒ (X = Δ(XT, P, T) = XT_i.Handler ≠ ⊥). Противоречие.

Из произвольности P и T следует, что DT ≈ XT.

3. Пусть $Del_{n,X}^{-1}$ — преобразование, обратное Del_n, удаляющему интервал X под номером n, такой, что UTF(X) = 1. Это преобразование эквивалентное в силу доказанного выше. Пусть $Sep_{n,C}^{-1}$ — преобразование, обратное $Sep_{n,C}$. Оно является эквивалентным, если C = XT_n.End = XT_{n+1}.Start. Пусть Int = <XT_n.End, XT_{n+1}.Start, XT_n.Type, XT_n.Handler>. Тогда UTF(Int) = 1, т.к. [XT_n.End, XT_{n+1}.Start) ⊂ NRF. Тогда эквивалентным является следующее преобразование:

$Concat_n = Sep_{n,(XT\_n+1).StartPC}^{-1}(Sep_{n,XT\_n.EndPC}^{-1}(Del_{n+1,Int}^{-1}))$. ■

Доказательство утверждения 2

Подтвердим оценку трудоемкости алгоритма. Трудоемкость первого этапа (сортировка) равна O(|XT|*log(|XT|)). Трудоемкость второго этапа равна O(|XT|), т.к. для каждого интервала верно, что он может попасть на стек только один раз и только один раз может быть оттуда снят, при этом на каждом шаге алгоритма осуществляется снятие одного интервала со стека. Таким образом, временная сложность алгоритма составляет O(|XT|*log(|XT|)). Емкостная сложность алгоритма составляет O(|XT|) т.к. это максимальный размер, которого может достигать стек. Докажем теперь, что алгоритм действительно проверяет, является ли таблица регулярной в смысле определения, данного в разделе 5.1.

Пусть есть интервалы A = XT_i и B = XT[j], пересекающиеся друг с другом без вложения или неправильно вложенные, причем i минимальное, а j минимальное для i. В любом случае A и B имеют непустое пересечение. Без ограничения общности предполагаем, что A.Start <= B.Start. Тогда в результате сортировки A находится в отсортированном массиве раньше B. Таким образом, A будет рассматриваться на втором шаге алгоритма раньше, чем B и, к моменту рассмотрения B будет еще находиться на стеке.

Действительно, если это не так, то между рассмотрениями A и B был рассмотрен некоторый интервал XT_k (i < k < j), такой что A не пересекался с XT_k, т.к. только в этом случае он мог быть снят со стека. В силу сортировки массива XT это означает, что XT_k.Start >= A.End. Также в силу сортировки массива XT, XT_k.Start <= B.Start. A.End <= XT_k.Start <= B.Start — противоречие с непустой пересеченностью A и B. Таким образом, действительно A находится на стеке в момент рассмотрения B.

Если A находится на стеке, то все элементы, находящиеся выше A, вложены в него, причем правильно вложены, т.к. их добавление не вытолкнуло A со стека, что означает, что они пересекались с A, а минимальность нерегулярности (A – B) гарантирует, что они не остановили алгоритм.

Когда B перестанет очищать стек, то на верхушке стека будет находиться некоторый XT_k (i <= k < j). При этом XT_k и B имеют непустое пересечение. Если k = i, то нерегулярность (A – B) определяется сразу же.

Пусть i < k. Если B неправильно вложен в A (т.е. в оригинальной таблице B имел больший номер, чем A), и при этом он имеет непустое пересечение с XT_k, то это означает и то, что он неправильно вложен в XT_k (или пересекается без вложения с ним), так как в силу правильного вложения XT_k в A, номер в оригинальной таблице XT_k меньше номера в оригинальной таблице A. Если же B пересекается без вложения с A, то, в силу вложения XT_k в A и пересечения B и XT_k, B пересекается без вложения с XT_k.

Таким образом, B пересекается без вложения или неправильно вложен в XT_k, находящийся на верхушке стека, что и будет определено алгоритмом. ■

Доказательство утверждения о корректности общего алгоритма декомпиляции.

Для шагов 1–3 общего алгоритма уже доказано, что таблица исключений, преобразованная этими шагами, эквивалентна оригинальной таблице исключений и что таблица исключений, полученная после 3-ого шага – регулярная.

Таким образом, надо доказать для шагов 4–5, что семантика обработки исключений для таблицы, поданной на вход 4-ому шагу, совпадает с семантикой обработки исключений в структуре TCB, получаемой на выходе.

Рассмотрим все возможные пути обработки исключений.

1. Отсутствие исключений.

В таком случае в структуре TCB выполнится try-блок. Затем, если finally-блок был скомпилирован jsr/ret инструкциями, то произойдет переход на операторы, в которые была декомпилирована инструкция jsr. Если finally-блок был скомпилирован открытой подстановкой, то он исполнится как декомпилированный код, следующий за структурой

ТСВ. Если же finally-блока не было, то исполнение также перейдет на операторы, следующие за ТСВ, что эквивалентно исполнению на оригинальном байткоде.

2. Обработка исключения.

Исполнение try-блока прервется на моменте возникновения исключения. Условие обработки исключения структурой ТСВ состоит из условия вложения инструкции, вызвавшей исключение, в try-блок этой структуры и условия наличия среди обработчиков catches обработчика, тип которого совместим по присваиванию с типом возникшего исключения. Все структуры, вложенные в рассматриваемую нами, не обработали исключение. Это означает, что рассматриваемая нами структура — первая подходящая по условию обработки исключения. Таким образом, из условия правильной вложенности всех интервалов рассматриваемая структура ТСВ соответствует первой записи таблицы исключений, подходящей под те же самые условия. В результате обработки исключения исполнение перейдет на инструкцию с адресом Handler, что эквивалентно обработке исключений в оригинальном байткоде. Finally-блок также будет исполнен, независимо от того, каким инструментом он был представлен в байткоде: jsr/ret или открытая подстановка.

3. Пропуск исключения.

Если в оригинальном байткоде предполагалось, что у блока обработки исключений есть finally-блок , то среди обработчиков обязательно появляется обработчик с нулевым значением CatchType, что означает произвольный тип исключения. Этот обработчик содержит finally-блок (реализованный jsr/ret или открытой подстановкой) и инструкцию throw, перевыбрасывающую исключение и тем самым продолжающую его обработку. Таким образом, пропуск исключения структурой — это частный случай обработки исключения. При этом пропущенное исключение будет обработано (или также пропущено) объемлющей структурой. Рассматриваемая же структура также из свойства правильной вложенности соответствует в таблице исключений записи с меньшим номером, чем объемлющая, что эквивалентно обработке исключений в оригинальном байткоде. ■

UDK: 004.4'422

Title: Exception handling blocks building at Java-bytecode decompilation

Author(s):

Solovyov Vladimir Valeryevich (A.P. Ershov Institute of Informatics Systems, Novosibirsk State University),

Lipsky Nikita Valeryevich (A.P. Ershov Institute of Informatics Systems)

Annotation: Java bytecode decompilation is a process of reverse translation that restores Java source code by the corresponding bytecode. Java bytecode is an intermediate representation based on abstract stack machine. It may have arbitrary control flow graph, whereas the Java language contains control structures that always form a strict hierarchy. Decompilation aims at restoring all control structures, including Java exception handling blocks. In the Excelsior RVM (Java Virtual Machine with a static compiler), bytecode is decompiled in a structural intermediate representation for further optimization. When building exception handling blocks, the Excelsior RVM's compiler assumes that the bytecode must be emitted by the standard Java source to bytecode compiler and uses a few heuristics to make reverse transformation. However, it is not always possible if the bytecode is produced by other instruments. This paper presents a decompilation algorithm that produces exception handling blocks given any correct bytecode. The algorithm has been implemented, integrated into the Excelsior RVM and tested on real-world applications.

Keywords: compilers, decompilers, Java, Java bytecode, exceptions handling, exceptions handling blocks, exception tables

УДК: 519.95

Название: Некоторые модели анализа и прогнозирования временных рядов

Автор(ы):

Шевченко И.В. (Институт систем информатики им А.П. Ершова СО РАН),

Аннотация: В статье рассматриваются несколько популярных классических моделей анализа и прогнозирования временных рядов. Вначале описываются относительно простые модели усреднения и сглаживания, затем модели авторегрессии, скользящего среднего, а также «смешанная» модель авторегрессии-скользящего среднего, полученная путем скрещивания двух последних моделей. Последней рассматривается интегрированная модель авторегрессии-скользящего среднего для случая нестационарных временных рядов.

Ключевые слова: прогнозирование, временной ряд, усреднение, экспоненциальное сглаживание, модель авторегрессии, скользящее среднее

1. Введение. Задача прогнозирования неопределенного будущего всегда была актуальна во многих областях. В данном случае, говоря о прогнозировании, мы подразумеваем анализ и прогнозирование *временных рядов* – наборов данных, которые были собраны или зафиксированы через последовательные промежутки времени. Существует огромное множество методов для выполнения этой задачи – начиная от простого экспоненциального сглаживания и заканчивая нейронными сетями. Многие методики стали неотъемлемой частью таких областей, как, например, эконометрика. Суть этих методов, если говорить в общем, состоит в подборе моделей, эффективно описывающих данные и способных быть продолженными в будущее.

Необходимость в методах прогнозирования обуславливается тем, что человек, обладая поразительными аналитическими способностями, а также знаниями и интуицией, склонен привносить в свои прогнозы некоторую степень субъективизма и недооценивать те или иные факторы [8].

Ниже рассматриваются наиболее популярные классические методы прогнозирования. Наиболее значимыми характеристиками при выборе той или иной модели прогнозирования являются, прежде всего, модель данных, на которые она ориентирована, и временная отдаленность выдаваемых ею прогнозов. Поэтому выбору модели предшествует анализ общей структуры ряда – чему и посвящен материал, предшествующий описаниям конкретных моделей прогнозирования.

2. Автокорреляция и частная автокорреляция. Главным отличием временного ряда от случайной последовательности является тот факт, что его члены являются взаимозависимыми. Степень связи значений двух случайных величин может быть выражена

коэффициентом корреляции между ними. Соответственно, когда мы хотим исследовать ряд на связь между его последовательными членами, разнесенными во времени на один и более периодов, мы аналогично можем вычислить коэффициент корреляции. В данном случае логичнее называть такие коэффициенты – *автокорреляциями* [4].

Иными словами *автокорреляция* – корреляция между величиной и ее запаздыванием в один и более периодов времени. Число периодов, по которым рассчитывается коэффициент автокорреляции, часто называют *лагом* или порядком автокорреляции.

Следующая формула показывает, как вычисляется коэффициент автокорреляции r_k между наблюдениями Y_t и Y_{t-1}, – т.е. с запаздыванием на k периодов:

$$r_k = \frac{\sum_{t=k+1}^{n}(Y_t - \bar{Y})(Y_{t-k} - \bar{Y})}{\sum_{t=1}^{n}(Y_t - \bar{Y})^2}$$

где

r_k – коэффициент автокорреляции с запаздыванием на k периодов;

$\bar{Y}$ – среднее значение ряда;

Y_t – наблюдение (отклик) в момент времени t;

Частная автокорреляция за промежуток времени k – это корреляция между Y_t и Y_{t-k}, т.е. отклик для периодов t и t-k после устранения влияния промежуточных значений Y_{t-1}, Y_{t-2}, … , Y_{t-k+1}.

Коррелограммой, или *автокорреляционной функцией*, является график коэффициентов автокорреляции для различных запаздываний во времени для заданного временного ряда. Анализ автокорреляционной функции и коррелограммы позволяет определить лаг, при котором автокорреляция наиболее высокая, а, следовательно, и лаг, при котором связь между текущим и предыдущими уровнями ряда наиболее тесная, т.е. с помощью анализа автокорреляционной функции и коррелограммы можно выявить структуру ряда.

Использование коэффициентов автокорреляции помогает в изучении временного ряда, давая ответ на основные интересующие нас вопросы: являются ли данные случайными, является ли ряд стационарным, или напротив, имеются ли в нем сезонные колебания?

И так, при анализе структуры временного ряда первый разумный вопрос, который должен возникать – есть ли в данных вообще какие-либо зависимости, является ли он случайным? Иными словами, нам необходимо выяснить есть ли зависимость между последовательными членами ряда. Анализ автокорреляций в данном случае может нам помочь. Если временной ряд имеет случайную природу, то коэффициенты автокорреляции для любого лага будут близки к нулю.

Следующий вопрос – имеют ли данные *тренд*? Тренд – долгосрочная компонента, отражающая возрастание или убывание временного ряда в течении длительного периода времени [7]. На рис. 1 можно наблюдать пример трендовой последовательности. Временной ряд, имеющий тренд, также называют *нестационарным*. Если ряд имеет тренд, то существует заметная тенденция в последовательности его членов. Соответственно, автокорреляции такого ряда будут иметь убывающий к нулю вид. Зачастую для анализа нестационарных рядов из них различными способами предварительно удаляется трендовая составляющая. Для этого, например, можно прибегнуть к взятию разности ряда, т.е. вместо исходного ряда

$$Y_0, Y_1, Y_2, Y_3, \ldots$$

рассматривать ряд вида:

$$Y_1 - Y_0, Y_2 - Y_1, Y_3 - Y_2 \ldots$$

Временные ряды, имеющие трендовую компоненту, также могут иметь *циклическую* компоненту – волнообразные флуктуации вокруг тренда.

Стационарным рядом называется ряд, основные статистические характеристики которого, такие как среднее значение и дисперсия, остаются постоянными во времени [7]. Таким образом, стационарный ряд колеблется вокруг некоторого фиксированного уровня или в канале (рис. 2). Можно заметить, что ряд, имеющий тренд, не является стационарным. Коэффициенты автокорреляции стационарного ряда убывают достаточно быстро.

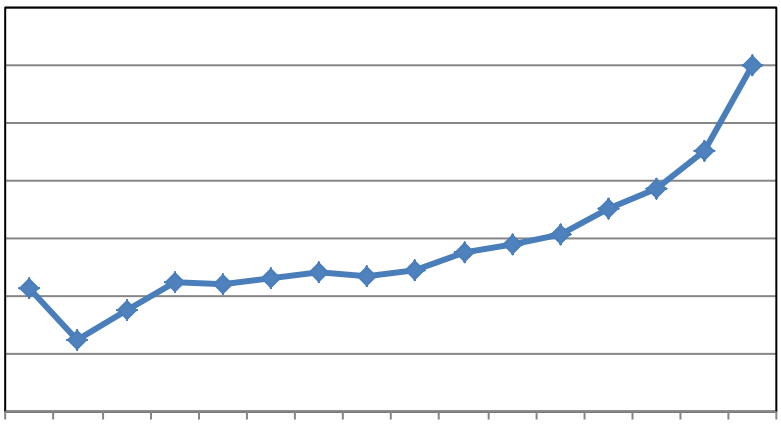


Рис. 1. Рид с трендовой составляющей

Если данные имеют некоторый периодичный, повторяющийся характер, то говорят, что в них проявляется *сезонная* компонента. Таким образом, сезонной компонентой называют периодические изменения в данных, повторяющиеся, например, из года в год. Это должно отражаться в виде значительных коэффициентов автокорреляции. Например, если ряд имеет годовую сезонность, и, скажем, соответствующие месяцы каждого года очень похожи, значит, стоит ожидать больших значений для автокорреляций с запаздыванием в 12 периодов.

3. Измерение ошибки прогноза. Ошибка прогноза есть мера отклонения прогноза от реального значения некоторого члена ряда. Если обозначить прогноз значения ряда Y_t в

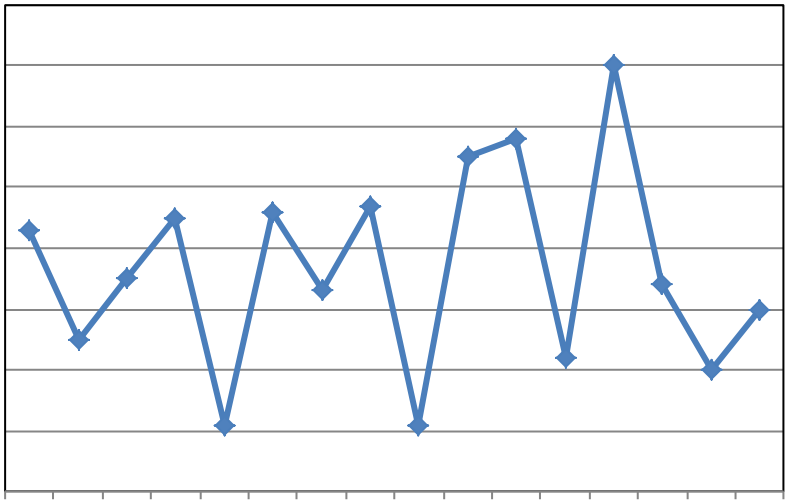


Рис. 2. Стационарный ряд

момент времени t за $\hat{Y}_t$, то нас будет интересовать обычно средняя оценка расхождений $\hat{Y}_t$ от величины Y_t. При этом естественным будет брать значение ошибки в каждый конкретный момент времени за $Y_t - \hat{Y}$.

Существуют разные методики оценки прогноза. Например, простое усреднение абсолютной величины отклонения:

$$E = \frac{1}{n}\sum_{t=1}^{n}|Y_t - \hat{Y}|$$

где E – ошибка прогноза. Суть данной оценки состоит в том, чтобы измерить среднюю величину отклонения в тех же величинах, что и исходные значения ряда.

Если же абсолютные значения ряда нам не так важны или мало о чем говорят, бывает полезно смотреть на относительное отклонение прогноза, т.е. отклонения относительно

абсолютных значений ряда. Для этого необходимо просто разделить предыдущую оценку на значение ряда в момент времени t:

$$E = \frac{1}{n}\sum_{t=1}^{n}\frac{|Y_t - \hat{Y}|}{Y_t}$$

можно также ошибку прогноза для каждого наблюдения определять в процентах по модулю:

$$E = \frac{1}{n}\sum_{t=1}^{n}\frac{|Y_t - \hat{Y}|}{Y_t} \times 100$$

Следующий довольно распространенный способ оценки прогноза – *среднеквадратичная ошибка*.

$$E = \frac{1}{n}\sum_{t=1}^{n}(Y_t - \hat{Y})^2$$

Особенность ее заключается в том, что методы, дающие более стабильную (равномерную) ошибку прогноза, будут иметь лучшую оценку, чем те, что имеют редкие, но значительные отклонения в прогнозе.

Существует множество аналогичных способов измерять ошибку прогноза. Выбор конкретной оценки зависит от конкретных задач.

4. Наивные модели. По названию можно догадаться, что это один из самых простых классов моделей. Суть методов заключается в предположении, что будущее лучше всего описывается самыми свежими данными [7]. Самый простой пример – предсказание типа «завтра будет так же, как сегодня», если обозначить прогнозируемую величину за $\hat{Y}_{t+1}$:

$$\hat{Y}_{t+1} = Y_t$$

Естественно предположить, что проблема данной модели в том, что случайные флуктуации сильно портят прогноз. Но, тем не менее, в условиях, например, недостатка исторических данных для анализа трудно придумать иную альтернативу.

Улучшить прогноз в данном случае может помочь изучение структуры ряда. Если, например, есть предположение, что в данных имеется трендовая составляющая, можно строить прогноз по следующей формуле [7]:

$$\hat{Y}_{t+1} = Y_t + \Delta Y_t$$

где

$$\Delta Y_t = Y_t - Y_{t-1}$$

Далее по мере накопления данных вместо последней разности ряда ΔY_t можно вычислять усредненное изменение ряда за один период. По аналогии можно построить более сложные модели.

5. Простые средние. Для анализа временных рядов часто используются методики усреднения и сглаживания, призванные убрать различные флуктуации и шумы, мешающие анализу. Методы усреднения и, в частности, простые средние помогают делать прогноз, основываясь на усредненных значениях прошлых наблюдений [7].

Очевидно, что данные ряда можно сгладить различными способами. При этом неизвестными параметрами может быть, например, количество последних наблюдений, которые берутся для прогноза, или весовые коэффициенты, сопоставленные каждому из них. В общем случае для оценки количества параметров и их конкретных значений практически во всех методах идут путем подгонки модели к некоторым данным предыстории. Затем параметры проверяются и уточняются по мере поступления новых данных [3]. Говоря о различных параметрах, необходимо учитывать не только достигаемую при их использовании точность прогноза, но и степень сложности получившейся модели.

Ниже приведена формула построения прогноза с помощью нахождения среднего значения ряда:

$$\widehat{Y}_{t+1} = \frac{1}{t}\sum_{i=0}^{t} Y_i$$

Подобные прогнозы приемлемы в случае стационарного ряда, когда процессы, порождающие этот ряд, уже стабилизировались.

6. Скользящие средние. В отличие от метода простых средних, где усреднялись все известные члены ряда, в методе скользящих средних используется только некоторое количество самых последних наблюдений. Соответственно, при поступлении новых данных они включаются в усреднение, а такое же количество самых «старых» наблюдений исключается.

Формула прогноза на основе скользящего среднего порядка n имеет следующий вид [7]:

$$\widehat{Y}_{t+1} = \frac{1}{n}\sum_{i=0}^{n-1} Y_{t-i}$$

Иными словами, скользящее среднее порядка n есть среднее арифметическое последних n наблюдений. Следует отметить, что в подобных моделях неизбежно проявляется эффект «запаздывания», когда кривая скользящей средней не успевает реагировать на быстрые изменения направления ряда. Степень запаздывания зависит от периода усреднения – чем

больше период, тем выше шансы запоздалой реакции на резкие движения. Таким образом, если в прогнозируемом ряде преобладают резкие движения, следует подбирать период скользящей средней как можно меньшим.

Если известно, что внутри интервала сглаживания имеется нелинейная тенденция, целесообразно применение взвешенных скользящих средних [3].

Главные достоинства данного метода – простота и наглядность.

7. Методы экспоненциального сглаживания. В целях дальнейшего улучшение точности прогноза в моделях, основанных на усреднении и сглаживании, целесообразно применение весовых коэффициентов, которые сопоставляются предшествующим членам временного ряда. Существуют также различные методы нахождения оптимальных весовых коэффициентов, которые позволяют добиваться улучшения прогнозов, адаптируясь к некоторым особенностям ряда. Во многих случаях используются последовательности весовых коэффициентов вида $w_i = \alpha^i$ где $\alpha < 1$. Это позволяет добиться того, что наиболее свежие данные будут иметь наибольший вклад в формируемый прогноз. Для модели экспоненциально взвешенного скользящего среднего существуют методики регулировки скорости затухания α. Так, в тех ситуациях, когда ошибка прогноза близка к нулю, скорость затухания α может быть увеличена, и наоборот [3].

Для уточнения прогноза обычно руководствуются принципом обратной связи, когда для корректировки используются ошибки в старых прогнозах. Таким образом, достигается постоянное обновление модели.

Формально процедуру экспоненциального сглаживания можно записать в следующем виде:

$$\hat{Y}_{t+1} = \alpha Y_t + (1-\alpha)\hat{Y}_t$$

или, переписав данное соотношение, получим

$$\hat{Y}_{t+1} = \hat{Y}_t + \alpha(Y_t - \hat{Y}_t)$$

где

$\hat{Y}_{t+1}$ – прогнозируемое значение на следующий период;

α – постоянная сглаживания ($0 < \alpha < 1$);

Y_t – наблюдение за текущий период t;

$\hat{Y}_t$ – прежний прогноз на период t;

Таким образом, экспоненциально сглаживание есть старый прогноз, скорректированный на ошибку старого прогноза с учетом весового коэффициента [7]. Экспоненциальное

сглаживание требует даже меньше арифметических операций, чем скользящие средние, а массив хранимой прошлой информации уменьшен до одного.

Также стоит отметить, что возможны различные стратегии выбора начального приближения. Так, можно, например, в качестве первого сглаживающего члена выбрать просто первое наблюдение или же взять некоторое усреднение первых наблюдений.

Главное достоинство модели прогнозирования, основанной на скользящей средней, заключается в том, что она способна адаптироваться к новому уровню процесса без значительных реакций на случайные отклонения. Однако данная модель дает значительную ошибку в случае, когда ряд имеет трендовую составляющую. Специально для этого случая существует несколько адаптивных моделей экспоненциального сглаживания. Например, *двухпараметрическая модель Хольта* [5]. В этом методе учитывается локальный линейный тренд, присутствующий во временных рядах. Идея метода состоит в том, что, если в данных присутствует локальный тренд, то, кроме оценки текущего уровня, необходимо оценивать также величину наклона. При этом постоянных сглаживания используется уже две. Это обеспечивает гибкость модели.

Прогноз на p периодов вперед, оценка уровня и тренда по модели Хольта описываются следующими выражениями, соответственно:

$$\widehat{Y}_{t+p} = L_t + pT_t$$

$$L_t = \alpha Y_t + (1-\alpha)(L_{t-1} - T_{t-1})$$

$$T_t = \beta(L_t - L_{t-1}) + (1-\beta)T_{t-1}$$

где

L_t – новая сглаженная величина;

α – постоянная сглаживания для данных ($0 < \alpha < 1$);

Y_t – наблюдение за текущий период t;

β – постоянная сглаживания для оценки тренда ($0 < \beta < 1$);

T_t – оценка тренда;

p – количество периодов вперед;

$\widehat{Y}_{t+p}$ – прогноз на p периодов вперед;

Дальнейшее улучшение модели Хольта разработал в 1960 году Винтерс. Его подход заключался в том, чтобы учесть влияние сезонных колебаний. Естественно, бесплатных улучшений не бывает, за них приходится платить возрастающей сложностью модели. В данном случае для учета сезонных колебаний добавляется еще одно уравнение и,

соответственно, еще один параметр – коэффициент сезонности. В результате мы получаем *трехпараметрическую модель Винтерса* [7]. Она задается следующей системой равенств:

$$L_t = \alpha \frac{Y_t}{S_{t-s}} + (1-\alpha)(L_{t-1} + T_{t-1})$$

$$T_t = \beta(L_t - L_{t-1}) + (1-\beta)T_{t-1}$$

$$S_t = \gamma \frac{Y_t}{L_t} + (1-\gamma)S_{t-s}$$

$$\widehat{Y}_{t+p} = (L_t + pT_t)S_{t-s+p}$$

где

L_t – новая сглаженная величина;

α – постоянная сглаживания для данных;

Y_t – наблюдение за текущий период t;

β – постоянная сглаживания для оценки тренда;

T_t – оценка тренда;

β – постоянная сглаживания для оценки сезонности;

S_t – оценка сезонности;

s – длительность периода сезонного колебания;

p – количество периодов вперед;

$\widehat{Y}_{t+p}$ – прогноз на p периодов вперед;

Как видно из приведенных уравнений, данная модель является расширением модели Хольта. Величина S_t призвана как раз учесть влияние сезонных колебаний и используется в последнем равенстве для корректировки прогноза. Что касается начальных значений, можно, например, взять первое значение сглаживания равным первому наблюдению, оценку тренда нулевой, а оценку сезонности единичной [7].

8. Авторегрессионные модели. Авторегрессионная модель порядка p имеет вид:

$$Y_t = \phi_0 + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} + \varepsilon_t$$

где

Y_t – значение временного ряда в момент времени t;

ϕ_i – оцениваемые коэффициенты;

ε_t – ошибка, накапливающаяся от неучтенных переменных;

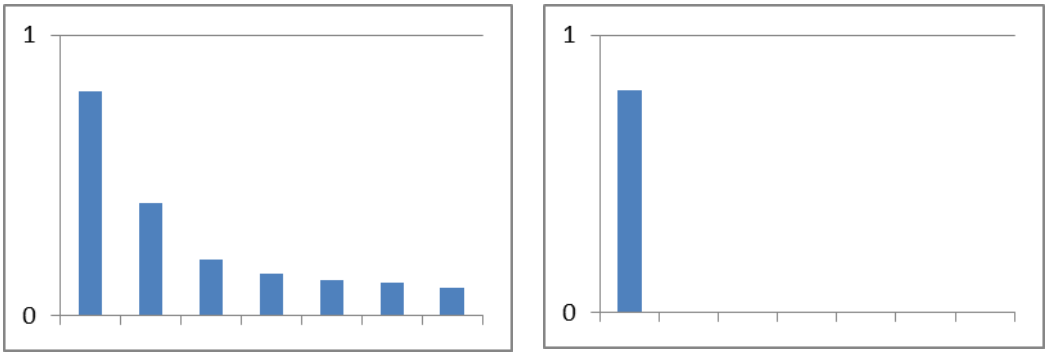
Свое название авторегрессионная модель получила ввиду того, что имеет вид регрессионной модели и использует в качестве независимой переменной запаздывающие

значения зависимой переменной. Такие модели применимы для стационарных временных рядов, при этом коэффициент ϕ_0 зависит от постоянного уровня ряда μ следующим образом:

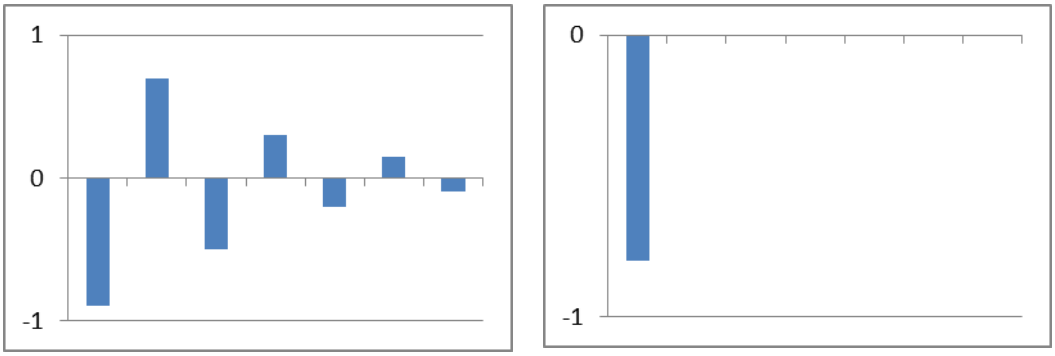
$$\phi_0 = \mu(1 - \phi_1 - \phi_2 - \cdots - \phi_p)$$

Изначально порядок процесса авторегрессии, приемлемо описывающий наблюденный ряд, для нас может быть неизвестен. Для его определения используется анализ частной автокорреляционной функции, который основан на том, что хотя процесс авторегрессии имеет бесконечно протяженную функцию автокорреляции, тем не менее, он может быть описан при помощи p ненулевых функций от автокорреляций. А именно, для процесса авторегрессии порядка p частная автокорреляционная функция обращается в ноль при задержке, превышающей p [2].

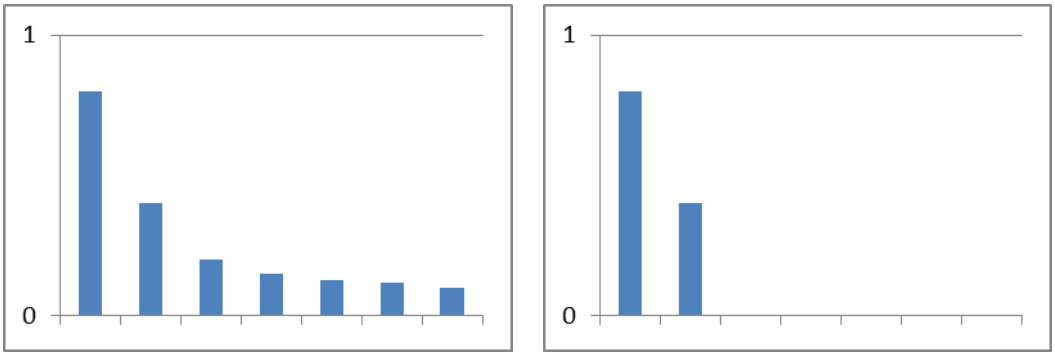
Автокорреляционные коэффициенты модели AR первого (а, б) и второго (в, г) порядка показаны на рис. 3. Можно заметить, что автокорреляционные коэффициенты модели AR имеют тенденцию затухать, стремясь к нулю, а частные автокорреляционные коэффициенты обращаются в ноль через промежуток времени, превышающий порядок модели [7]. В общем случае автокорреляционная функция стационарного процесса авторегрессии состоит из совокупности затухающих экспонент и затухающих синусоид [2].



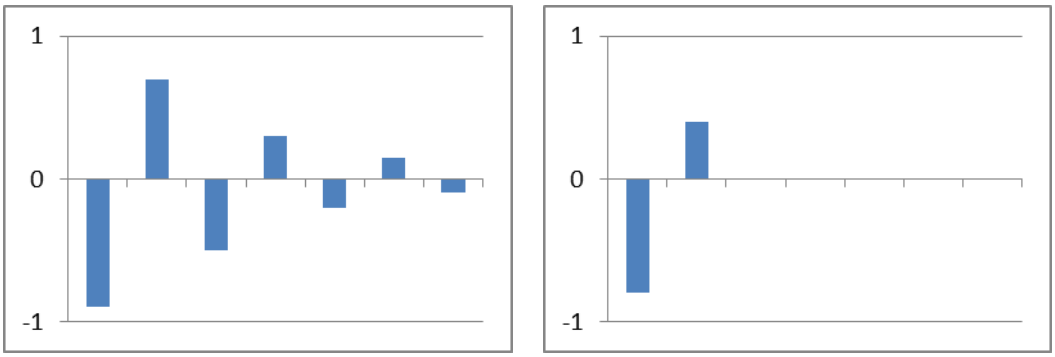
а)



б)



в)



г)

Рис. 3. Коэффициенты автокорреляции и частной автокорреляции моделей авторегрессии первого и второго порядка

9. Модели со скользящим средним. Модель со скользящим средним порядка q имеет следующий вид:

$$Y_t = \mu + \varepsilon_t - \omega_1\varepsilon_{t-1} - \omega_2\varepsilon_{t-2} - \cdots - \omega_p\varepsilon_{t-q}$$

где

Y_t – значение временного ряда в момент времени t;

μ – постоянное среднее ряда;

ω_i – оцениваемые коэффициенты;

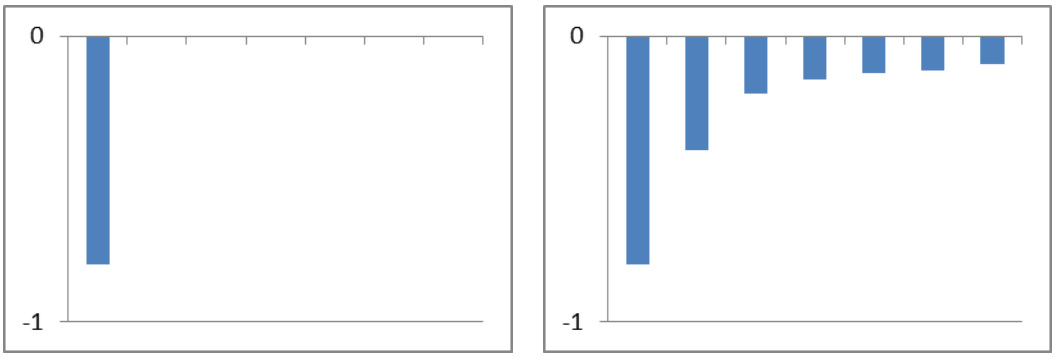
ε_t – ошибки в предыдущие моменты времени, которые в момент t включены в Y_t;

Данное уравнение отличается от уравнения авторегрессии тем, что Y_t зависит от предыдущих значений ошибок, а не от значений отклика. Таким образом, модели со скользящим средним дают прогноз функции Y_t на основе линейной комбинации прошлых ошибок, а не предыдущих значений самого ряда. Нужно также заметить, что в данном случае мы не налагаем каких-то строгих ограничений на коэффициенты ω_i – они не должны давать в сумме единицу, как и иметь положительный знак.

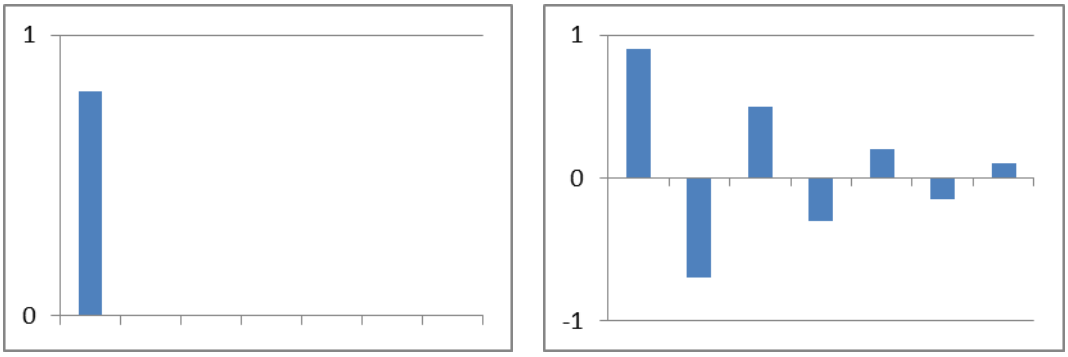
Что касается термина «скользящее среднее», упомянутого в названии модели, это скорее исторически сложившееся название, не имеющее никакого отношения к процедуре вычисления скользящего среднего на основе усреднения всех или части последних наблюдений ряда [7].

Значение модели со скользящим средним заданного порядка удобно получать последовательным добавлением прошлых ошибок, которые были включены в прогноз прошлых наблюдений. На рис. 4 представлено теоретическое поведение коэффициентов автокорреляции и частичной автокорреляции модели со скользящим средним первого (а, б) и второго (в, г) порядка. Сравнив их с аналогичными графиками для модели авторегрессии, можно заметить, что в данном случае поведение коэффициентов совсем иное. Отличие в том, что у модели со скользящим средним коэффициенты автокорреляции обращаются в нуль сразу после первого (MA(1)) и второго (MA(2)) периода, в то время как частные автокорреляции стремятся к нулю постепенно [1].

Мы видим, что коэффициенты автокорреляции для модели MA(q) равны нулю при запаздывании на период, превышающий порядок модели. Это важное свойство можно использовать при выборе порядка по экспериментальным данным.



а)



б)

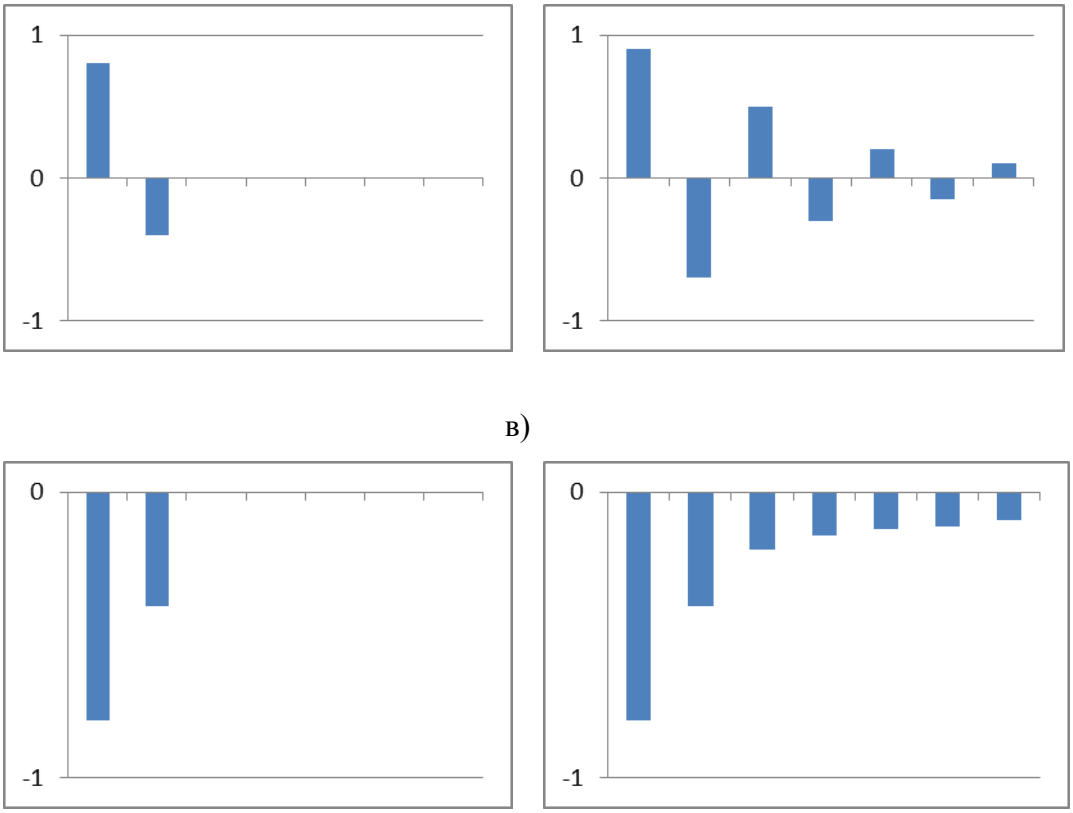


Рис. 4. Коэффициенты автокорреляции и частной автокорреляции моделей скользящего среднего первого и второго порядка

10. Модель авторегрессии-скользящего среднего. Ввиду того, что в представлении временного ряда присутствует некий дуализм, а именно – один и тот же ряд может быть представлен двумя моделями, то сама собой напрашивается мысль «смешать» эти две модели, получив, таким образом, модель авторегрессии-скользящего-среднего. Если мы возьмем p членов в авторегрессионной части и q членов в части модели со скользящим средним, получившуюся смешанную модель обозначают ARMA(p, q). Модель имеет следующий вид [7]:

$$Y_t = \phi_0 + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} + \varepsilon_t - \omega_1 \varepsilon_{t-1} - \omega_2 \varepsilon_{t-2} - \cdots - \omega_p \varepsilon_{t-q}$$

Таким образом, получившаяся модель зависит уже и от предыдущих значений отклика, и от предыдущих ошибок. Модель может быть использована для описания широкого спектра поведений стационарных временных рядов.

Что касается теоретических значений коэффициентов автокорреляции и частной автокорреляции, нетрудно догадаться, что для смешанной модели и те и другие будут плавно затухать, не обрываясь на каком-то шаге.

Количество членов для каждой модели – AR члены и MA члены – определяется видом функции автокорреляции и частной автокорреляции конкретного ряда. Данный подход является частью методологии Бокса-Дженкинса по выбору наиболее подходящей модели [6]. Также в этом процессе необходимо учитывать так называемые критерии выбора модели. На практике результаты работы модели ухудшаются при попытке увеличить количество авторегрессионных членов или скользящего среднего, – это так называемая проблема «перепараметризации». Поэтому принято начинать подбор с наименьшего числа этих членов, при необходимости добавляя их постепенно. Соответственно, нужно стараться делать количество членов модели как можно меньшим [7].

11. Интегрированная модель авторегрессии-скользящего среднего. Многие эмпирические временные ряды (например, цены на бирже) ведут себя так, будто они не имеют фиксированного среднего. Но даже при этом в их поведении наблюдается некоторая однородность – любая часть временного ряда по своему поведению во многом подобна любой другой, если их привести к одному уровню. До этого момента мы говорили о построении модели стационарного ряда. На практике в первую очередь необходимо проверить ряд на стационарность, – т.е. изменяются ли его значения в окрестности некоторого неизменного уровня. Если мы делаем вывод о стационарности ряда, тогда мы можем использовать одну из разобранных моделей. В противном случае можно попытаться превратить нестационарный ряд в стационарный. Обычно это делается путем взятия разностей. В этом случае вместо исходного ряда Y_t мы будем строить модель для ряда ΔY_t:

$$\Delta Y_t = Y_t - Y_{t-1}$$

В некоторых случаях данную процедуру придется повторить некоторое количество раз – обычно не больше двух-трех:

$$\Delta^2 Y_t = \Delta(\Delta Y_t) = Y_t - 2Y_{t-1} - Y_{t-2}$$

Еще один распространенный прием – взятие логарифма разности.

Таким образом, если разность некоторого порядка есть стационарный смешанный процесс авторегрессии-скользящего среднего, мы получаем модель для нестационарных временных рядов – *интегрированная модель авторегрессии-скользящего среднего* [2]. Если обозначить количество применения оператора разности как d, полученную модель можно обозначить стандартной записью ARIMA(p, d , q), где p – порядок авторегрессии, q – порядок членов скользящего среднего. Соответственно, при обнулении одного или двух параметров мы можем получать модели ARMA(p, q), AR(p), MA(q).

12. Критерии выбора модели. Как уже упоминалось ранее, процедура подбора модели ARMA основана на анализе коэффициентов автокорреляции и частной автокорреляции с теоретическими показателями, присущими моделям авторегрессии и скользящего среднего. Однако эта процедура неизбежно вносит некую долю субъективизма в выбор подходящей модели. К тому же нередки ситуации, когда две модели отвечают нужной структуре данных. В таком случае необходимо смотреть на среднеквадратичную ошибку модели, а также на количество на ее порядки.

Исходя из таких рассуждений, были разработаны несколько критериев, призванных помочь нам в процедуре подбора модели. Одним из таких критериев является информационный критерий Акаике (обозначается AIC), дающий числовую оценку пригодности модели (чем меньше числовое значение – тем лучше). Критерий имеет следующую формулу [6]:

$$AIC = \ln \sigma^2 + \frac{2}{n} r$$

где

$\ln$ – натуральный логарифм;

$\mathbf{y}^2$ – остаточная сумма квадратов, деленая на количество наблюдений;

n – количество наблюдений;

r – общее количество слагаемых в модели ARIMA

Другой распространенный критерий – Байесовский информационный критерий (обозначается BIC) вычисляется похожим образом [7]:

$$AIC = \ln \sigma^2 + \frac{\ln n}{n} r$$

Байесовский критерий более чувствителен к количеству параметров по сравнению с предыдущим критерием. Но на практике оба критерия зачастую дают похожие результаты.

13. Заключение. В статье рассмотрены несколько популярных классических моделей, предназначенных для анализа и прогнозирования временных рядов. Были рассмотрены модели усреднения и сглаживания, имеющие свои достоинства, такие как относительную вычислительную простоту и наглядность. Упомянуты сопутствующие подзадачи подбора и оптимизации параметров моделей, а также прилагающиеся к этим процедурам опасности, связанные с растущей сложностью модели прогнозирования. Мы видели, как из простых моделей авторегрессии и скользящего среднего, описывающих стационарные временные ряды, получается «смешанная» модель ARMA, которая с помощью преобразования временного ряда посредством разностного оператора распространялась на нестационарные

временные ряды. Суть процедуры подбора подходящей модели сводилась к выбору порядка каждой составляющей путем анализа структуры функций автокорреляции и частной автокорреляции анализируемого ряда. Необходимо отметить, что как раз процедура изучения поведения коэффициентов корреляции и частной автокорреляции является зачастую трудно формализуемой и требует наличия определенных навыков. Рассмотренные критерии выбора модели, Акаике и Байесовский информационный критерии, не всегда способны помочь в выборе наилучшей модели. Хотя на практике при необходимости прогонки больших тестов, требующих полной автоматизации процесса, зачастую практикуется выбор наилучших моделей по информационному критерию.

Список литературы

1. Айвазян С.А., Мхитарян В.С. Прикладная статистика и основы эконометрики. М.: ЮНИТИ, 1998. 1000 с.
2. Бокс Дж., Дженкинс Г. Анализ временных рядов. Прогноз и управление. М.: Мир, 1974. Т. 1,2. 601 с.
3. Грешилов А. А., Стакун В. А., Стакун А. А. Математические методы построения прогнозов. М.: Радио и связь, 1997. 112 с.
4. Елисеева И.И. Эконометрика, 2-е издание. М: Финансы и статистика, 2004. 344 с.
5. Лукашин Ю. П. Адаптивные методы краткосрочного прогнозирования временных рядов. М.: Финансы и статистика, 2003. 416 с.
6. Магнус Я.Р., Катышев П.К., Пересецкий А.А. Эконометрика. Начальный курс. М.: Дело, 2004. 576 с.
7. Ханк Д.Э., Уичерн Д.У., Райтс А.Дж. Бизнес-прогнозирование, 7-е издание. М.: Вильямс, 2003. 656 с.
8. Makridakis S., Wheelwright S.C., Hyndman R.J. Forecasting: Methods and Applications, 3rd edition. New York: Wiley, 1998. 656 p.

UDK: 519.95

Title: Some models of time series analysis and prediction

Author(s):

Igor V. Shevchenko (A.P. Ershov Institute of Informatics Systems)

Abstract: This paper presents some popular classical methods used for time series analysis and prediction. At first relatively simple models of averaging and smoothing are described, then autoregressive and moving average models, and a “mixed” autoregressive-moving-average model as a result of crossing of the latter two mentioned models. At last an autoregressive integrated moving average model is described.

Keywords: forecasting, time series, averaging, exponential smoothing, autoregressive model, moving average

УДК: 519.688

Название: Разработка программы построения и кластеризации геномных профилей с использованием GPU

Автор(ы):

Никитин С.И. (Новосибирский государственный университет),

Черемушкин Е.С. (Институт систем информатики им. А. П. Ершова СО РАН)

Аннотация: В процессе считывания РНК на определенных участках ДНК — сайтах связывания формируется комплекс белков, называемых транскрипционными факторами. Этот комплекс позволяет закрепиться РНК-полимеразе и начать считывание РНК. Задача поиска сайтов связывания на ДНК является сложной ввиду наличия многих факторов, влияющих на связывание. В их числе — наличие других сайтов связывания на небольшом расстоянии от рассматриваемого сайта. Для исследования этой зависимости авторами были введены в рассмотрение гистограммы распределения плотности сайтов на геноме, названные геномными профилями.

В рамках данной работы реализован алгоритм предсказания сайтов связывания с помощью весовых матриц, написана его параллельная реализация для архитектуры NVidia CUDA, реализован алгоритм построения геномных профилей, алгоритмы иерархической кластеризации и кластеризации К-средних для геномных профилей. Реализован алгоритм, позволяющий строить случайные иерархии транскрипционных факторов на основании существующей биологической классификации для того, чтобы оценить качество полученной классификации геномных профилей. Соответствующая программа написана на языке C++ и предназначена для быстрого построения геномных профилей и их первичного анализа. Проведен анализ сходства классификации геномных профилей с биологической классификацией транскрипционных факторов для исследования влияния взаимного расположения сайтов связывания на ДНК.

Ключевые слова: РНК, ДНК, геном, весовые матрицы, транскрипционные факторы, сайты связывания с факторами транскрипции, кластеризация, программная система, Nvidia CUDA

1. Введение. Появление новых экспериментальных технологий в областях, связанных с обработкой генетической информации, позволяющих с высокой эффективностью исследовать молекулярно-генетические системы и процессы, стимулирует развитие биоинформатики. Задачи определения однонуклеотидных полиморфизмов и даже задачи полной расшифровки генома становятся доступными не только крупным институтам и консорциумам, но и небольшим организациям. В результате в последнее время в

молекулярной биологии и генетике произошел информационный взрыв, сопровождаемый огромным ростом объемов экспериментальных данных[1]. В частности, в ходе работ была получена ценная информация о генном и геномном уровнях организации жизни. Но надежды на то, что с развитием технологии секвенирования мгновенно возникнет четкая картина, описывающая все процессы, происходящие в клетке, пока не оправдываются. Все больше и больше ощущается недостаток алгоритмов и программ, которые бы позволили выделить из этого моря данных биологически значимую, структурированную информацию.

Все больший интерес вызывают такие технологии, как микрочипы (microarray analysis), Chip-on-Chip и Chip-seq[2]. Микрочиповый эксперимент позволяет измерить экспрессию (количество произведенной РНК) практически для всех генов в клетке одновременно. Этот профиль экспрессии различен в клетках, которые выполняют разные функции. Если клетка стала по каким-то причинам функционировать неправильно, например, в результате заболевания или под действием определенных веществ, то это отразится на профиле экспрессии. Таким образом, в теории с помощью таких экспериментов возможно определить массу зависимостей между генами, а также сделать количественные оценки. Chip-on-Chip и Chip-Seq эксперименты позволяют найти набор фрагментов ДНК, к которым присоединяются транскрипционные факторы, что (по плану создателей) позволит определить, как регулируется экспрессия генов. В результате выяснилось, что in vitro (в пробирке), связывание происходит не так, как in vivo (в жизни): транскрипционный фактор может связаться с определенным «сайтом» in vitro, но in vivo этого связывания не будет, т. к. ДНК не будет освобождена от нуклеосомной структуры на этом участке, либо другие транскрипционные факторы будут связаны с этим участком, что помешает связыванию, и т.д.

Таким образом, несмотря на рост количества информации, не теряет актуальности создание и улучшение высокопроизводительных информационно-компьютерных систем и технологий, предназначенных для анализа и интерпретации экспериментальных данных о связывании транскрипционных факторов с ДНК. Эти программы являются необходимыми для решения фундаментальных задач биологии, а тем более — для практического использования в биомедицине и биотехнологии.

Несмотря на то, что биоинформатика является очень молодой наукой, в ней уже существуют свои традиционные направления, такие как компьютерный анализ ДНК, РНК и белковых последовательностей, распознавание функциональных сайтов, реконструкция пространственных структур биополимеров, теоретический и компьютерный анализ

структурно-функциональной организации геномов и белков. Одним из исследуемых в биоинформатике процессов является процесс транскрипции и связанные с ним задачи.

Транскрипцией называется синтез рибонуклеиновой кислоты (РНК) происходящий при непосредственном участии дезоксирибонуклеиновой кислоты (ДНК) [3]. Транскрипция — один из фундаментальных биологических процессов, происходящий в живых клетках, первый этап реализации генетической информации, записанной в ДНК в виде линейной последовательности 4 типов мономерных звеньев — нуклеотидов. Этот процесс осуществляется специальными ферментами — ДНК зависимыми РНК-полимерами. В результате образуется цепь РНК, последовательность звеньев которой повторяет последовательность звеньев одной из двух цепей копируемого участка ДНК. Продуктом транскрипции являются 4 типа РНК, выполняющих различные функции:

- информационные, или матричные, РНК, выполняющие роль матриц при синтезе белка рибосомами (трансляция);
- рибосомальные РНК, являющиеся структурными компонентами рибосом;
- транспортные РНК, являющиеся основными элементами, осуществляющими доставку аминокислот к месту синтеза белка;
- РНК, играющие роль активаторов репликации ДНК.

Транскрипция ДНК происходит отдельными участками, в которые входит один или несколько генов. Фермент РНК-полимераза определяет начало такого участка (промотор), присоединяется к нему, расплетает двойную спираль ДНК и копирует, начиная с этого места, одну из её цепей, перемещаясь вдоль ДНК и последовательно присоединяя мономерные звенья — нуклеотиды — к образующейся РНК в соответствии с принципом комплементарности (т. е. напротив аденина присоединяется урацил, напротив гуанина присоединяется цитозин, и наоборот. Однако в ДНК отсутствует урацил, таким образом, аденин присоединяется напротив тимина). По мере движения РНК-полимеразы растущая цепь РНК отходит от матрицы, и двойная спираль ДНК позади фермента восстанавливается. Когда РНК-полимераза достигает конца копируемого участка (терминатора), РНК отделяется от матрицы. Число копий разных участков ДНК зависит от потребности клеток в соответствующих белках и может меняться в зависимости от условий среды или в ходе развития организма. Это достигается за счет регуляции. Понимание механизма регуляции экспрессии генов — важнейшая задача биологии. При изучении регуляции экспрессии на уровне транскрипции важно не только определить белки-регуляторы (транскрипционные факторы), но и участки их связывания с последовательностью ДНК, а также условия, при которых происходит связывание на данном участке. В настоящее время в открытом доступе

находится большое количество секвенированных геномов и данных по экспрессии генов, что позволяет изучать регуляцию путем анализа последовательностей с помощью вычислительных методов. Задача поиска регуляторных мотивов в наборе последовательностей ДНК — классическая задача биоинформатики. К настоящему моменту создано огромное количество алгоритмов поиска мотивов, однако все они имеют свои ограничения, и не существует универсального алгоритма, который решал бы эту задачу.

2. Моделирование процесса транскрипции. Инициация транскрипции является сложным процессом, протекание которого зависит от множества факторов. Если говорить об эукариотах, то такими факторами могут являться форма последовательности ДНК вблизи начала транскрибируемой области, а также в более удалённых участках, называемых энхансерами и сайленсерами. Дополнительно на процесс транскрипции влияет наличие или отсутствие различных белковых транскрипционных факторов [4]. Факторы транскрипции — белки, которые регулируют транскрипцию путем связывания со специфичными участками ДНК — сайтами связывания. Транскрипционные факторы выполняют свою функцию самостоятельно либо в комплексе с другими белками. Различают репрессорные и активирующие транскрипционные факторы, которые, соответственно, снижают или повышают константу связывания РНК-полимеразы с регуляторными последовательностями экспрессируемого гена [5]. Определяющая черта факторов транскрипции — наличие в их составе одного или более ДНК-связывающих доменов, которые взаимодействуют с сайтами связывания, расположенными в регуляторных областях генов. Транскрипционные факторы бывают конститутивные (всегда активные в клетке) и активируемые (активируются только при определенных условиях). Активируемые, в свою очередь, разделяют на тканеспецифические (активируются только в определенных тканях организма) и сигнал-зависимые, или рецепторы (требуют внешнего сигнала для активации).

Набор транскрипционных факторов, стимулирующих, в конечном счете, транскрипцию, является различным для разных типов генов. Присоединившись к соответствующему сайту связывания на ДНК, комплекс из транскрипционных факторов позволяет закрепиться РНК-полимеразе на старте транскрипции и произвести считывание РНК. Сайты связывания имеют длину в среднем 10-20 нуклеотидов и для одного и того же транскрипционного фактора имеют схожие последовательности. Это объясняется тем, что транскрипционные факторы имеют специфическую форму, которая позволяет им закрепляться на последовательностях определенного типа. Но, несмотря на кажущуюся простоту, определить, является ли данная последовательность сайтом, сложно. Это обусловлено тем, что на связывание влияют и другие факторы, в частности — другие сайты в окрестности исследуемого.

TRANSFAC (eukaryotic trans-acting Transcriptional regulatory Factors and cis-acting regulatory sites database) — база данных, содержащая информацию о сайтах связывания эукариотических факторов транскрипции в геномах и о связывающихся белках. Содержит экспериментально подтвержденные регуляторные сайты эукариот (от дрожжей до человека), а также белки, действующие как факторы транскрипции или вместе с ними [6].

Весовые матрицы (PWM – position weight matrix) библиотеки Transfac компании Biobase являются самым распространенным средством выявления потенциальных сайтов связывания с транскрипционными факторами на ДНК.

PWM, которые впервые были введены для характеристики сайтов инициации транскрипции и трансляции у E. coli (кишечная палочка) [7,8], хорошо подходят для описания сайтов связывания факторов транскрипции и способны количественно охарактеризовать частые и редкие вариации в последовательности сайтов. PWM представляют собой матрицу L × 4 (L — длина сайта), где номер строки соответствует позиции нуклеотида в сайте, а по столбцам стоят частоты встречаемости данного нуклеотида в данной позиции сайта.

№	A	C	G	T
01	13	4	4	3
02	14	3	5	2
03	2	7	1	14
04	23	0	0	1
05	24	0	0	0
06	1	23	0	0
07	0	1	17	6
08	1	1	21	1
09	11	5	4	4
10	12	5	2	5

Рис.1. Весовая матрица V$VMYB_01. По вертикали — позиция в сайте, по горизонтали нуклеотид. В ячейках частота встречаемости данного нуклеотида в данной позиции в наборе экспериментальных сайтов

Вес, порождаемый матрицей при выравнивании с данным участком последовательности, в нашем случае вычисляется как сумма элементов матрицы, соответствующих нуклеотидам, стоящим в каждой позиции рассматриваемого участка.

Обычно перед вычислением веса матрица конвертируется в скоровую матрицу. Это происходит с помощью домножения каждого элемента матрицы на информационный

коэффициент, соответствующий данной строке, либо подсчетом вероятностной матрицы и логарифмированием значений.

№	A	C	G	T	…
01	13	4	4	**3**	T
02	14	3	5	**2**	T
03	2	7	**1**	14	G
04	23	**0**	0	1	C
05	**24**	0	0	0	A
06	1	23	**0**	0	G
07	0	1	17	**6**	T
08	1	1	**21**	1	G
09	**11**	5	4	4	A
10	12	5	**2**	5	G
					…

Рис. 2. Вычисление веса последовательности TTGCAGTGAG с помощью весовой матрицы V$VMYB_01

Таким образом, PWM предоставляет достаточно полное описание участка ДНК, с которым способен связываться конкретный белок, и может быть применена при сканировании геномной последовательности для поиска сайтов, дающих достаточно хороший вес. Использование PWM позволяет достаточно эффективно предсказывать сайты связывания белков.

Однако следует отметить, что, несмотря на все свои достоинства, PWM все-таки имеет несколько недостатков. Одним из них является то, что PWM не учитывает взаимное влияние соседних позиций сайта, а только учитывает общую форму текущей подпоследовательности. Однако наличие таких зависимостей было показано для некоторых факторов [9,10]. Другим недостатком данного метода поиска потенциальных сайтов связывания является то, что он не учитывает присутствие других сайтов в окрестности текущего. Для решения этой проблемы было предложено исследовать гистограммы распределения плотности сайтов на геноме, названные геномными профилями. Такие профили строятся для каждой весовой матрицы и описывают распределение густоты сайтов связывания, т. е. показывают, сколько фрагментов генома расположены на последовательности ДНК с одной густотой сайтов, сколько с другой и т.д. Перед авторами была поставлена задача реализации алгоритма построения геномных профилей для весовых матриц и обоснования сходства классификации геномных профилей с

биологической классификацией транскрипционных факторов для исследования влияния взаимного расположения сайтов связывания на ДНК.

Алгоритм поиска потенциальных сайтов связывания с факторами транскрипции основан на вычислении веса рассматриваемой подпоследовательности генома и сравнении результата с некоторым наперед заданным порогом. Т.к. геномные последовательности имеют большую длину, процесс поиска занимает длительное время. Эта проблема также рассматривается в данной работе, поскольку алгоритм используется при построении геномных профилей весовых матриц.

2. Постановка задачи. Целью данной работы являлось создание программы, позволяющей для каждой весовой матрицы строить геномные профили, графически отображать полученные результаты в виде гистограмм распределения, обрабатывать полученные профили с помощью некоторых алгоритмов кластеризации; написать параллельную реализацию алгоритма поиска потенциальных сайтов связывания для GPU; написать алгоритм создания случайных деревьев по существующему дереву классификации транскрипционных факторов для исследования применимости геномных профилей. Сделать вывод о сходстве классификации геномных профилей с биологической классификацией транскрипционных факторов. В качестве языка программирования выбран C++ с использованием инструментария Qt, так как он позволяет упростить процесс создания приложений, обладает обширной документацией, а также является кроссплатформенным, что расширяет возможности использования программы на отличных от Windows платформах. Для параллельной реализации алгоритма поиска потенциальных сайтов связывания была выбрана технология NVidia CUDA, т.к. программно она основана на языке C, использует принципы SIMD архитектуры (одним набором инструкций независимо обрабатывается большой объем данных); к тому же существуют специализированные видеоадаптеры Tesla, отличающиеся от обычных видеоадаптеров качеством реализации, точностью расчетов и большей вычислительной мощностью.

3. Геномные профили. В настоящее время физические свойства многих транскрипционных факторов подробно изучены, благодаря большому количеству экспериментальной информации о сайтах связывания с транскрипционными факторами. Это позволило получить весовые матрицы, предсказывающие потенциальные сайты связывания на ДНК с большой точностью. Однако на связывание влияет не только форма факторов транскрипции, но и другие условия, такие как доступность сайта, наличие кофакторов, внутрицепочечные взаимодействия, влияние других сайтов связывания в окрестности данного и другие. Таким образом, затруднительно на основании последовательности генома

достоверно предсказать реальное место посадки транскрипционного фактора на ДНК in vivo. В связи с этим изучение условий, влияющих на связывание транскрипционных факторов и ДНК, является актуальной задачей. Для анализа влияния сайтов связывания, находящихся в окрестностях данного, нами было предложено ввести новую конструкцию «геномные профили». Геномные профили представляют собой гистограммы распределения плотности сайтов на геноме и характеризуют кучность расположения потенциальных сайтов связывания с факторами транскрипции. Пока рассматривается влияние сгруппированности сайтов одного типа, независимо от других сайтов.

Алгоритм построения геномных профилей для весовой матрицы состоит из 3 этапов:

1. Предсказание потенциальных сайтов связывания.

Пусть M — весовая матрица размера $4xN$. Зафиксируем некоторую позицию i в последовательности генома G и будем рассматривать его подпоследовательность длины N: $G(i, N)$. С помощью весовой матрицы вычислим вес выбранного отрезка генома w_i как сумму элементов матрицы, соответствующих нуклеотидам, стоящим в каждой позиции рассматриваемого участка, а затем нормируем полученное значение на отрезок $[0, \dots, 1]$ следующим образом:

$$w = \frac{(w_i - w_{min})}{(w_{max} - w_{min})}$$

где w_{min} и w_{max} — минимальный и максимальный вес последовательности

Полученный вес w сравнивается с некоторым наперед заданным порогом c. Если $w \geq c$, то сайт связывания в данном месте на последовательности считается распознанным, в противном случае нераспознанным.

2. Построение профиля распознанных сайтов.

Зафиксируем весовую матрицу M и порог c, с которым мы предполагаем сайт связывания распознанным. Разобьем последовательность генома на участки длины L (в нашем случае L=100000) нуклеотидов, в каждом из них последовательно проведем поиск потенциальных сайтов связывания. Таким образом, мы получим профиль распознанных сайтов, то есть количество $V_{c,M}(i)$ найденных сайтов для каждого участка.

В последовательной реализации на фиксированном участке поиск сайтов происходит последовательно от начала до конца отрезка ДНК. Параллельная реализация алгоритма заключается в следующем: зафиксируем участок длины L. Произведем поиск сайтов связывания для каждой позиции i с помощью отдельного потока на GPU. Благодаря возможности запускать сотни потоков одновременно происходит одновременная обработка

большого объема данных, что приводит к значительному увеличению скорости обработки участка.

3. Построение профиля матрицы.

Упорядочим полученный профиль распознанных сайтов $V_{c,M}(i)$ по возрастанию. Исключим нулевые элементы, т.к. они образуются на участках ДНК, заполненных полиNсигналом. Далее, отбросим 5% значений сверху и снизу, тем самым удаляя выбросы распределения $V_{c,M}(i)$. Полученное распределение $V'_{c,M}$ преобразуем db гистограмму следующим образом: найдем V'_{max} и V'_{min} — максимум и минимум $V'_{c,M}$. Разобьем отрезок $[V'_{min}, V'_{max}]$ на $T = 20$ равных фрагментов $d_1 \dots d_T$. Далее посчитаем количество $V'_{c,M}(i)$, попавших в каждый из d_t. Повторим процедуру для каждой хромосомы. Получим искомый геномный профиль $P_{c,M}(t)$.

3.1. Кластеризация. Кластерный анализ (англ. Data clustering) — задача разбиения заданной выборки объектов на подмножества, называемые кластерами, так, чтобы каждый кластер состоял из схожих объектов, а объекты разных кластеров существенно отличались. Рассматриваем в качестве элементов выборки геномные профили весовых матриц, которые по сути являются векторами одной длины. Соответственно, стоит задача разбиения набора векторов на подмножества близких по метрике элементов. Для реализации алгоритмов кластеризации выбраны две основные метрики пространства R^n:

- евклидова: $S(x,y) = \sqrt{\sum_{i=0}^{n}(x_i - y_i)^2}$;
- $S(x,y) = \sum_{i=0}^{n}|x_i - y_i|$.

В качестве методов кластеризации выбраны два известных в литературе и широко распространенных алгоритма: метод К-средних (K-means) и иерархический метод.

3.1.1. Метод К-средних.

1. Задать число K кластеров. Выбрать случайным образом K элементов, которые будут являться «центрами масс» кластером на начальном этапе.
2. Отнести каждый элемент к кластеру с ближайшим «центром масс».
3. Пересчитать «центры масс» согласно текущему составу элементов в кластерах. Координаты «центров масс» вычисляются как средние арифметические соответствующих координат элементов кластера.
4. Если состав кластеров не изменился, то стоп, иначе на шаг 2.

Итерация алгоритма имеет сложность $O(N * K)$, где N — число элементов выборки, K — число кластеров.

3.1.2. Иерархическая кластеризация. Метод иерархической кластеризации заключается в следующем:

1. Задать число K кластеров. Каждый элемент выборки является отдельным кластером. Таким образом, перед началом работы имеем N кластеров.
2. Выбираем два кластера, расстояние между которыми минимально, и объединяем их в один. Таким образом уменьшаем общее число кластеров. В качестве расстояния между кластерами можно выбрать один из следующих вариантов:
   - расстояние между двумя наиболее отдаленными элементами;
   - расстояние между двумя наиболее близкими элементами;
   - среднее расстояние между всеми парами объектов в них.
3. Если достигнуто необходимое число K кластеров, то стоп, иначе шаг 2.

Алгоритм имеет сложность $O(N^2)$.

4. Программная система для расчета геномных профилей. Для получения и обработки геномных профилей весовых матриц, была разработана программная система GenomeSignal. Программа написана на языке C++ с использованием кросс-платформенного инструментария разработки ПО Qt, а также с использованием технологии параллельных вычислений NVidia CUDA. Кроме этого, в программе использована технология параллельных вычисления для систем с общей памятью OpenMP. Тестирование проводилось на операционных системах Windows 7 x86/x64, Windows 8 x64.

4.1. Внутренне строение программной системы. В связи с тем, что C++ является объектно-ориентированным языком программирования, внутреннее устройство программы реализовано в виде классов. Взаимодействие между классами показано на рис. 3.

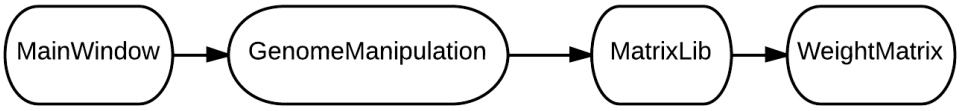


Рис.3. Взаимодействие классов в программе GenomeSignal

Класс **WeightMatrix** является хранилищем значений количеств предсказанных сайтов транскрипционных факторов для некоторой фиксированной весовой матрицы, содержит набор геномных профилей, соответствующих данной матрице, а также предоставляет процедуры для построения данных структур.

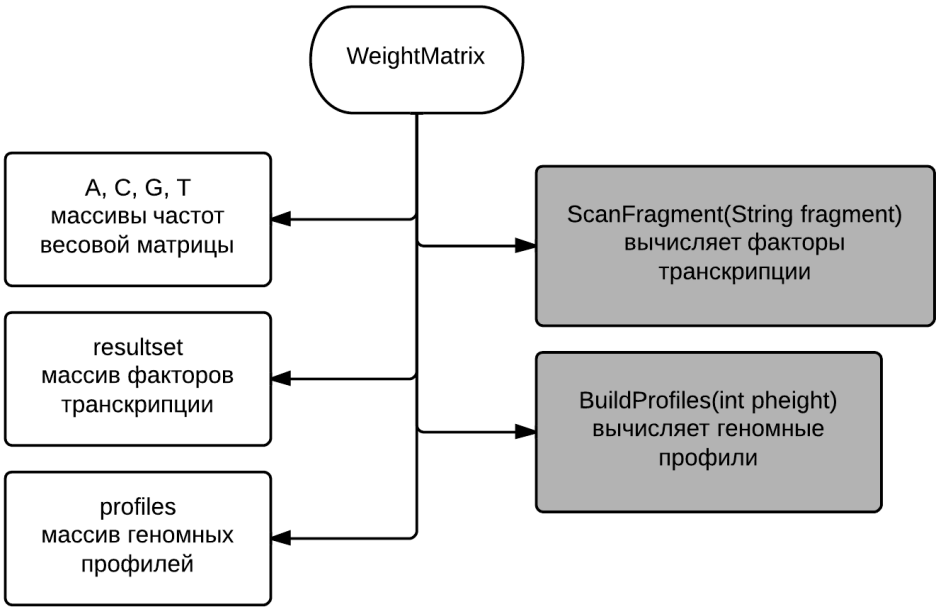


Рис.4. Основные поля и методы класса WeightMatrix

Класс **MatrixLib** является хранилищем результатов работы для всех весовых матриц, т.к. содержит массив элементов типа WeightMatrix для всех загруженных весовых матриц и общие методы получения профилей и предсказания факторов транскрипции.

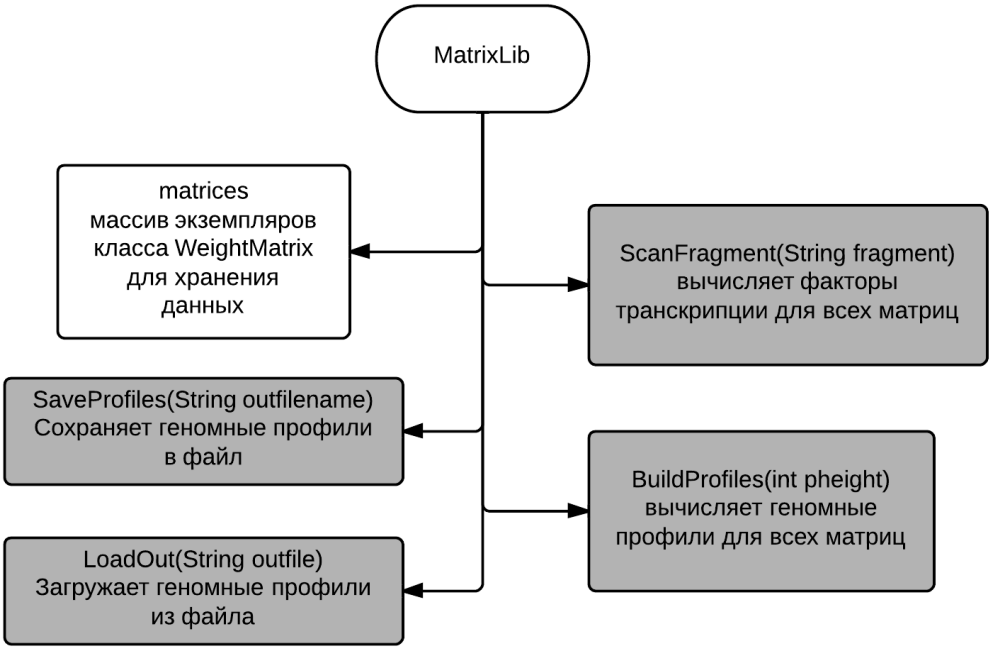


Рис.5. Основные поля и методы класса MatrixLib

Класс **GenomeManipulation** является основным классом программы GenomeSignal. Он включает в себя реализации всех алгоритмов для получения и обработки геномных профилей.

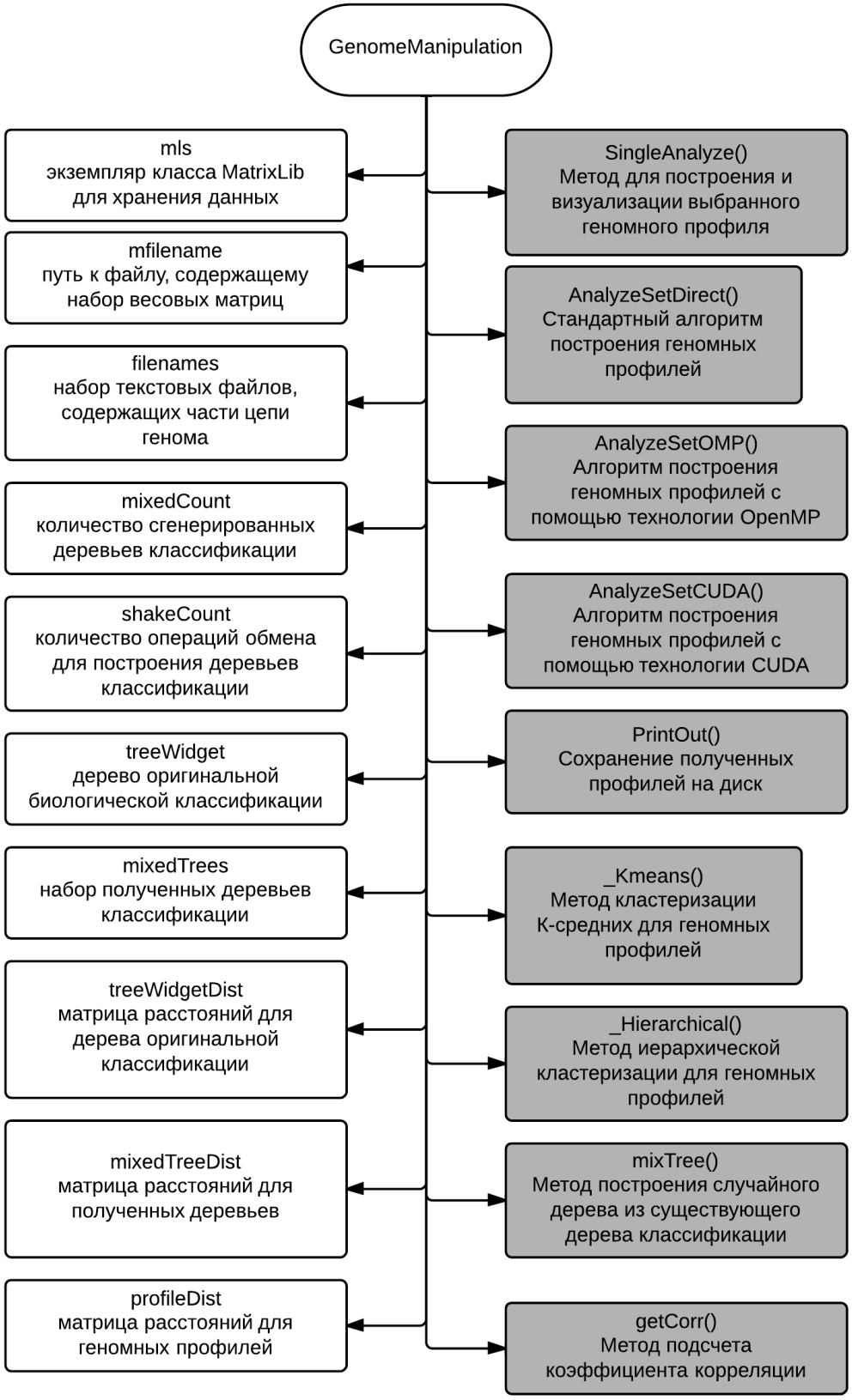


Рис.6. Основные поля и методы класса GenomeManipulation (параметры методов опущены для сокращения записи)

Класс **MainWindow** является классом взаимодействия программы с пользователем. В нем собраны методы работы интерфейса, реализованы возможности хранения последних путей к файлам, логирования последних произведенных вычислений, а также проверки системы на возможность использования тех или иных технологий. Для независимой работы в каждом режиме и во избежание потери выполненных вычислений, в программе используется 4 экземпляра класса GenomeManipulation.

4.2. Пользовательский интерфейс программной системы. Внешний вид программы представляет собой диалоговое окно, в котором вкладки «пакетная обработка данных», «визуализация геномных профилей», «кластеризация», «сравнение» отвечают соответствующим режимам работы.

4.2.1. Режим пакетной обработки данных. На рис. 7. программа GenomeSignal настроена в режим «пакетная обработка данных».

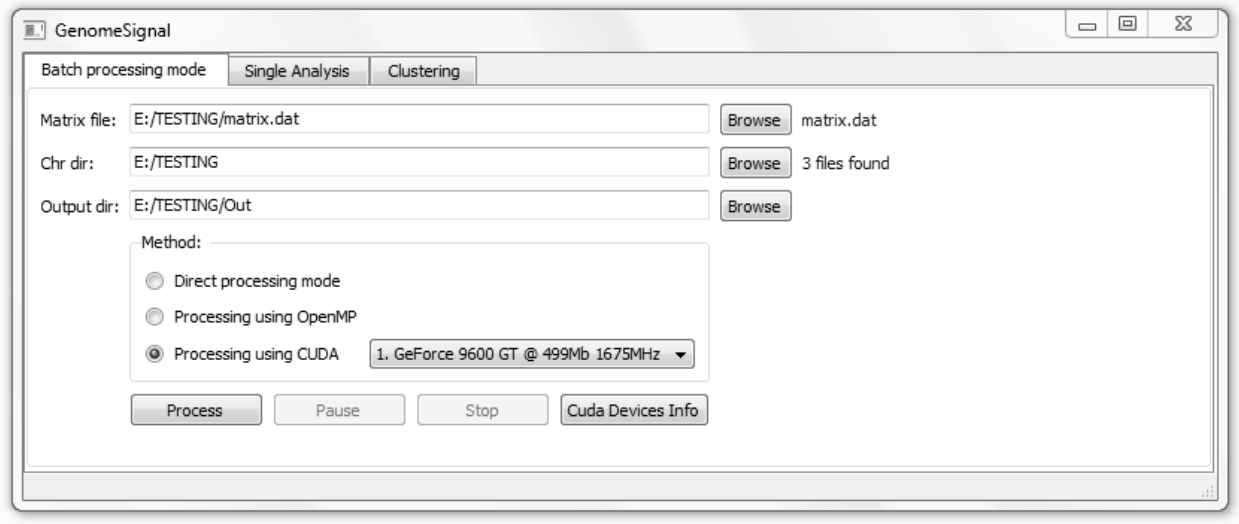


Рис.7. Режим пакетной обработки данных

Входными данными этого режима являются файл с библиотекой матриц, а также набор файлов с последовательностями генома. Пользователь может выбрать директорию, в которую будут записаны результаты работы. Программа обработает входные данные и сохранит полученные геномные профили с соответствующими именами в указанную директорию. Это длительная процедура, поэтому в программе реализована возможность останавливать процесс или ставить обработку на паузу. Также пользователь может выбрать один из вариантов работы программы: последовательная обработка, обработка с применением технологии OpenMP, обработка с применением технологии CUDA (если в

системе нет установленных совместимых с CUDA видеоадаптеров, данный метод будет недоступен).

4.2.2. Режим визуализации геномных профилей. На рис. 8. программа GenomeSignal настроена в режим «визуализация геномных профилей».

Рис.8. Режим визуализации геномных профилей

Входными данными этого режима является файл с библиотекой матриц, а также файл с последовательностью ДНК. Пользователь выбирает весовую матрицу и порог, для которых строится профиль. После обработки программа отображает результат в виде двух графиков. График справа (синий цвет) отображает распределение сайтов на данном участке ДНК. График слева (красный цвет) отображает соответствующий геномный профиль. Также пользователь может загрузить уже сохраненный в файле геномный профиль. В этом случае будет отображен только график геномного профиля.

Пользователь имеет возможность экспортировать полученный профиль в Microsoft Excel.

4.2.3. Режим кластеризации. На рис. 9 программа GenomeSignal настроена в режим «кластеризация».

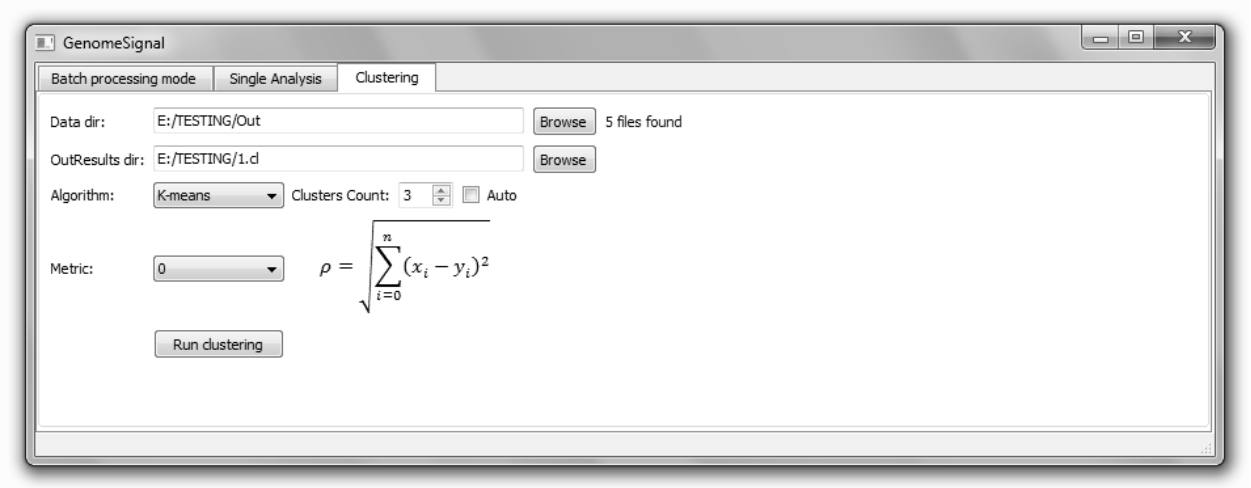


Рис.9. Режим кластеризации

Входными данными этого режима является набор файлов, содержащих построенные геномные профили. Пользователь выбирает метод кластеризации, количество кластеров, а также метрику, с помощью которой будут производиться расчеты. Программа обрабатывает данные и сохраняет результат разбиения в указанный пользователем файл.

4.2.4. Режим сравнения. На рис. 10 программа GenomeSignal настроена в режим «сравнение».

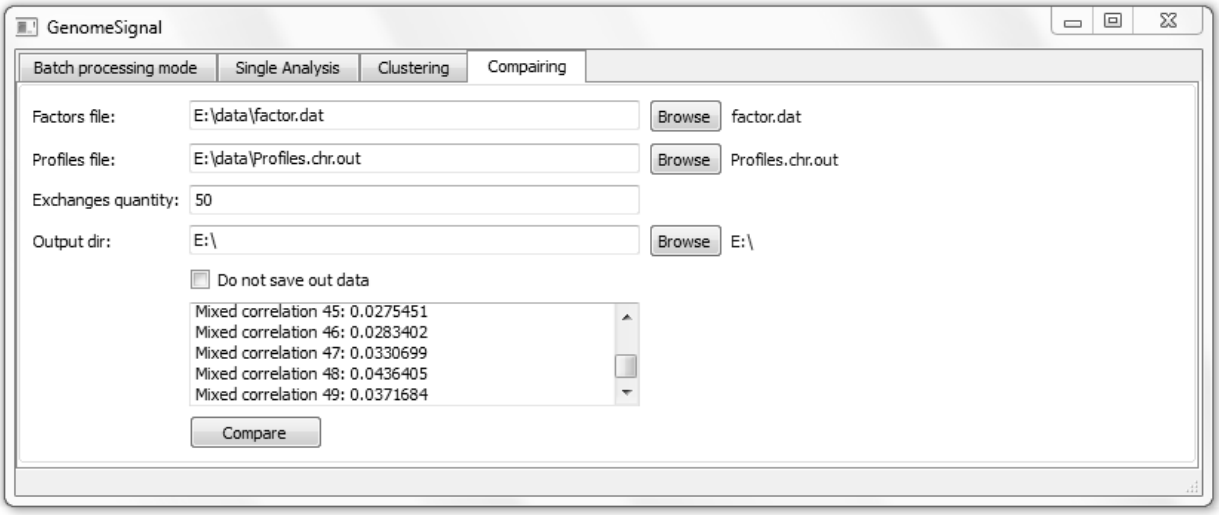


Рис.10. Режим сравнения

Входными данными этого режима являются файл с иерархией транскрипционных факторов, а также файл с геномными профилями. Пользователь может выбрать директорию, в которую будут записаны сгенерированные деревья и полученные матрицы расстояний, либо отказаться сохранять данные, выбрав соответствующее поле программы. Расстояние между весовыми матрицами соответствует расстоянию между их геномными профилями и

вычисляется как расстояние между векторами длины 20 с евклидовой метрикой. Иерархия транскрипционных факторов представляет собой дерево, в вершинах которого записаны соответствующие транскрипционным факторам весовые матрицы. Соответственно, в данном случае расстоянием между двумя весовыми матрицами является кратчайшее по количеству рёбер расстояние между ними в дереве иерархии. Результатом работы программы является набор коэффициентов корреляции между матрицей геномных профилей и матрицами иерархии факторов транскрипции (загруженной биологической иерархии и сгенерированных случайных иерархий).

5. Реализация на GPU. Развитие современной вычислительной техники предопределило два совершенно разных подхода к обработке данных, используемых в компьютерах в настоящее время. Еще 10 лет назад центральные процессоры испытывали стремительный рост быстродействия и параллельной обработки данных. Были введены специализированные векторные возможности (SSE2 и SSE3). Однако рост частот центральных процессоров резко замедлился из-за физических ограничений и высокого энергопотребления. К настоящему времени развитие ЦПУ сводится к размещению нескольких ядер на одном процессоре. Сейчас на рынке представлены центральные процессоры с количеством ядер до 16.

Противоположностью центральным процессорам выступают процессоры для видеокарт, которые изначально были спроектированы для параллельных векторных вычислений, используемых в 3D-графике. Современные видеочипы содержат сотни математических исполнительных блоков, выполняющих операции одновременно, что может значительно ускорить множество вычислительно интенсивных приложений. Для того чтобы задействовать возможности видеочипов для вычислений общего назначения, были разработаны технологии неграфических расчётов GPGPU (General-Purpose computation on GPUs). Наибольшее распространение на данный момент получила программно-аппаратная вычислительная архитектура CUDA от компании NVidia. По данным компании, около 20% мощности всего списка быстрейших суперкомпьютеров Top500 обеспечивают NVidia GPU [11].

Архитектура CUDA основана на расширении языка C и даёт возможность организации доступа к набору инструкций графического ускорителя и управления его памятью при организации параллельных вычислений. Архитектура видеочипов, изначально разработанных из идеи SIMD (одни инструкции для большого потока данных) вычислений, делает перспективным их использование для решения многих научных задач. Еще одним немаловажным преимуществом применения видеокарт для неграфических вычислений является применение более быстрой памяти, поэтому видеочипам доступна в разы большая

пропускная способность памяти, что играет важную роль при обработке большого потока данных. На данный момент существует множество вычислительных кластеров на GPU, построенных на основе специализированных видеоадаптеров Tesla от компании NVidia с поддержкой CUDA, а также возможна установка нескольких видеокарт в персональные компьютеры с помощью технологии SLI, что позволяет получать еще большую производительность путем использования нескольких видеочипов одновременно.

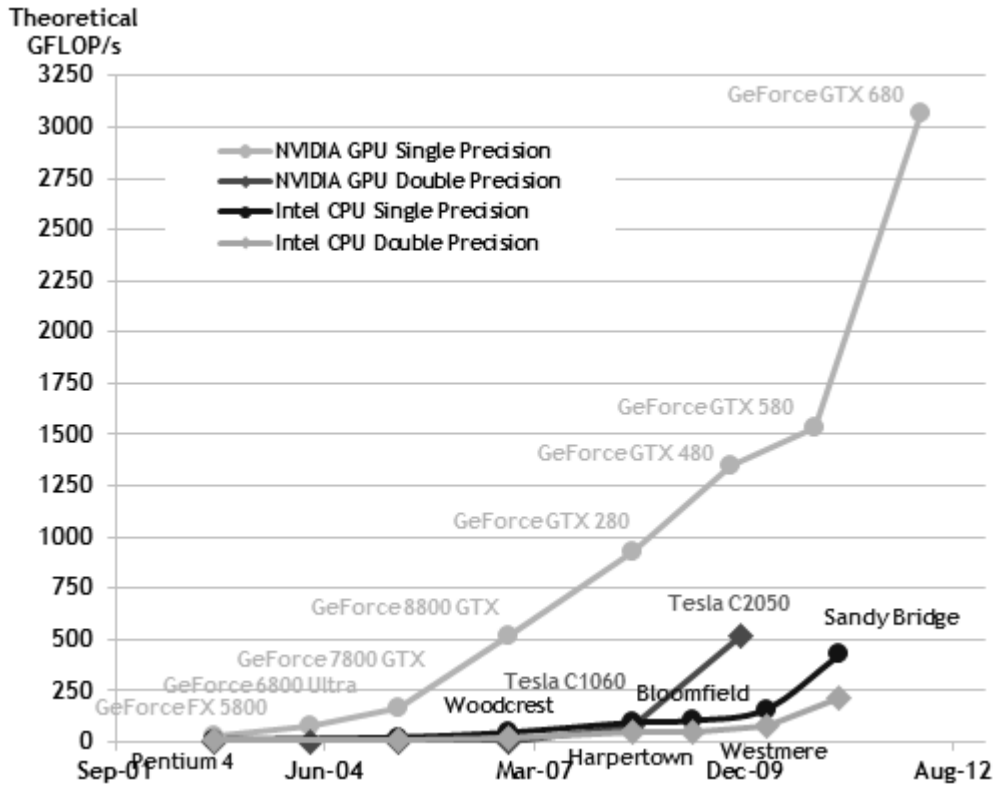


Рис. 11. Пиковая мощность для операций с плавающей точкой на CPU и на GPU [12]

В рамках нашей научной работы одной из подзадач является поиск потенциальных сайтов связывания с транскрипционными фактора на геноме. Данная подзадача является перспективной для параллельной реализации вычислений на видеочипе, так как требует обработки большого объема данных (геном) с помощью весовых матриц. Это позволяет ожидать многократного прироста производительности алгоритма по сравнению с его стандартной реализацией.

Блок-схема стандартного алгоритма поиска потенциальных сайтов связывания показана на рис. 12. Т.к. весовые матрицы имеют длину в среднем 10-20, мы имеем не меньше 5000 одинаковых операций вычисления веса участка генома для каждой весовой матрицы, которые выполняются последовательно. При параллельной реализации алгоритма мы

создаем сразу необходимое количество потоков, при этом создание и переключение между потоками на видеокарте происходит быстрее, чем на центральном процессоре. Каждый поток вычисляет вес участка со своим смещением, чем и достигается ускорение алгоритма.

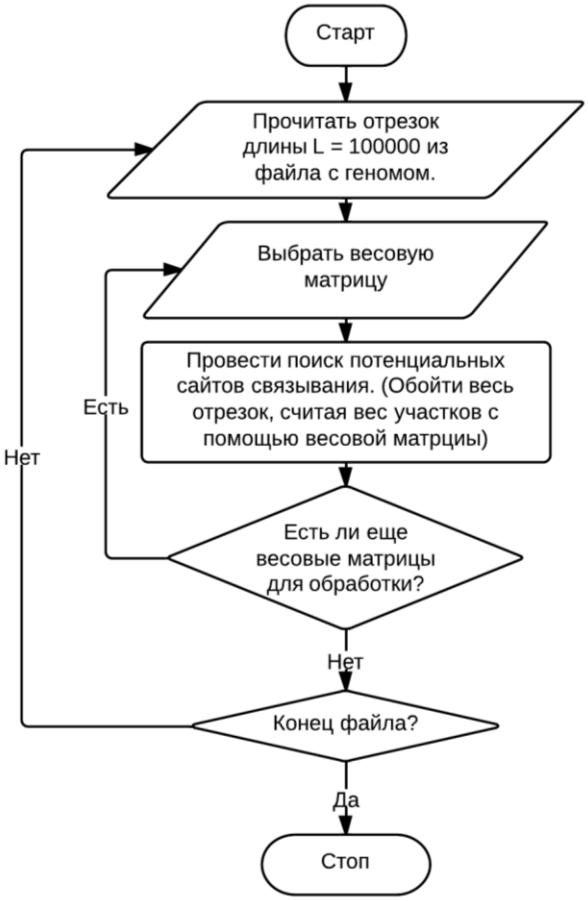


Рис.12. Алгоритм поиска потенциальных сайтов связывания

При достаточно большом количестве ядер CUDA все потоки произведут вычисления одновременно, в противном случае параллельно будет запущено максимально возможное количество потоков, остальные будут ожидать своей очереди.

Код на CUDA пишется в отдельном файле с расширением .cu, так как для его компиляции с основной программой требуется предварительная сборка с помощью CUDA компилятора nvcc. Заголовки функций, которые будут вызываться из основной программы, помещаются с заголовочный файл. Этот файл можно подключить в любом месте программы, как любой заголовочный файл. Синтаксис кода является расширением синтаксиса языка C. Для того чтобы работать с памятью на видеокарте, существуют функции **cudaMalloc** и **cudaFree.** Копирование из оперативной памяти в память видеокарты и обратно происходит с помощью функции **cudaMemcpy**, где последний параметр определяет направление копирования (**host** – оперативная память, **device** – видеопамять).

```
void ParallelStart(int *iA, int *iC, int *iG, int *iT, int *iDim, int *iW, char *iGene, int imcount, int *oRes)
{
    int Nthreads = 500;
    int Nblocks = 20;

    .......

    int *GlobRes = new int[imcount * 15];

    cudaMalloc((void**)&dlocRes, sizeof(int)*(Nblocks*Nthreads*15));
    for ( int i = 0; i < imcount; i++ )
    {
        for (int cl = 0; cl < Nblocks*Nthreads*15; cl++) hlocRes[cl] = 0;
        cudaMemcpy(dlocRes, hlocRes, sizeof(int)*(Nblocks*Nthreads*15), cudaMemcpyHostToDevice);
        SearchSites<<<Nblocks, Nthreads>>>(iA, iC, iG, iT, iDim, iW, iGene, i, dlocRes);
        cudaMemcpy(hlocRes, dlocRes, sizeof(int)*(Nblocks*Nthreads*15), cudaMemcpyDeviceToHost);

        .......

    }
    cudaFree( dlocRes );

    .......
    delete [] hlocRes;
    delete [] GlobRes;
}
```

Рис.13. Работа с памятью в CUDA программе

Запуск функции на видеочипе происходит с указанием двух параметров: числа блоков и числа потоков в каждом блоке (на рис. 13 функция **SearchSites** запускается на видеочипе). Соответственно, будет создано число потоков, равное произведению этих параметров. Во время выполнения можно определить, какой поток выполняет функцию, и использовать номер потока как параметр (на рис. 14 в **m_starter** помещается номер потока, в котором мы находимся).

```
__global__ void SearchSites(int *iA, int *iC, int *iG, int *iT, int *iDim, int *iW, char *iGene, int iMSel, int *oRes)
{
    int m_starter = blockIdx.x * 5000 + threadIdx.x * 10; // Вычисляем, в каком потоке мы находимся

    .......

if (denominator != 0)
{

    for ( int i = 0; i < 10; i++ )
    {
        record = 0;
        for ( int j = 0; j < iDim[m_matrIndex]; j++ )
        {
          if(( iGene[m_starter + i + j] == 'A' )||( iGene[m_starter + i + j] == 'a' )) record = record + iA[m_matrArrIndex + j];
          if(( iGene[m_starter + i + j] == 'C' )||( iGene[m_starter + i + j] == 'c' )) record = record + iC[m_matrArrIndex + j];
          if(( iGene[m_starter + i + j] == 'G' )||( iGene[m_starter + i + j] == 'g' )) record = record + iG[m_matrArrIndex + j];
          if(( iGene[m_starter + i + j] == 'T' )||( iGene[m_starter + i + j] == 't' )) record = record + iT[m_matrArrIndex + j];
        }
        record = record - iW[m_matrIndex * 2 + 0];
        for ( int coff = 15; coff > 0; coff--)
        {
            rt = 1.0 - (0.025*(float)coff);

            if ( (float)(record) >= rt*(float)(denominator) ) oRes[blockIdx.x*500*15 + threadIdx.x*15 + coff - 1]++;
            else break;
        }
    }
}

}
```

Рис.14. Функция, которую можно запустить на видеокарте

Как видно, процесс написания программы на CUDA слабо отличается от обычной программы на языке C. Это позволяет быстро адаптировать алгоритм для работы на видеокарте. Для нашей задачи поиска потенциальных сайтов связывания параллельная реализация алгоритма показала значительный прирост производительности. При этом тестирование было проведено также для видеокарт разных поколений —– GeForce 9600GT выпускается с 2008 года, построена на базе процессора G94, имеет 64 ядра CUDA; GeForce GTX650 выпускается с 2012 года, построена на базе процессора GK107, имеет 384 ядра CUDA. Результаты можно видеть на рис. 15.

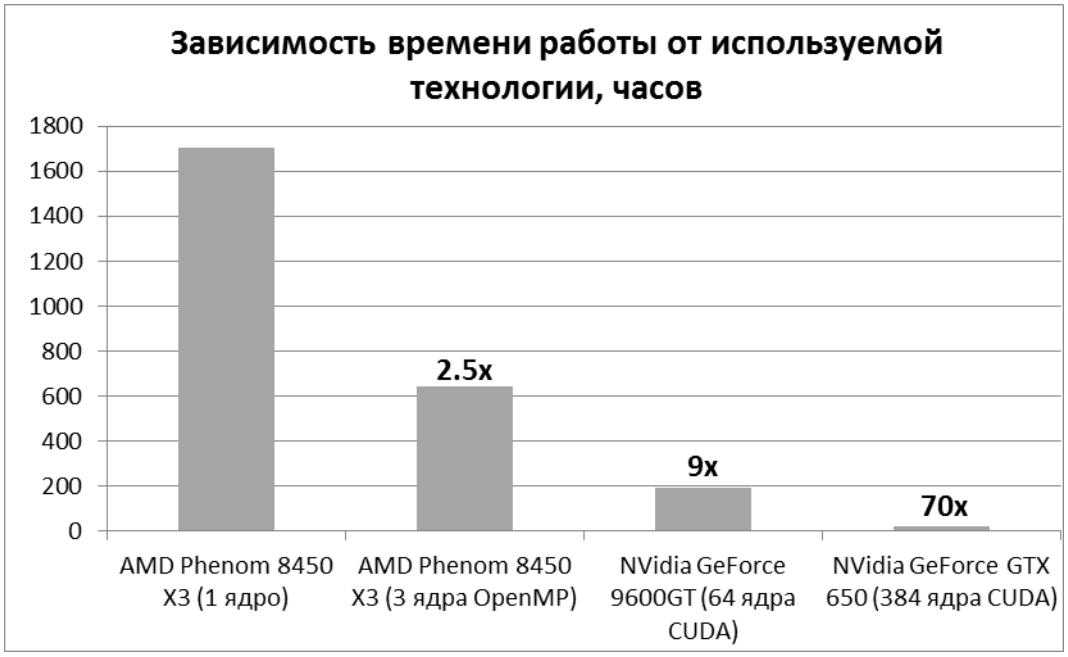


Рис.15. Работа алгоритма в разных режимах

Видно, что реализация алгоритма для работы на видеокарте позволила получить скачек в производительности до 70 раз. Можно предположить, что на более мощных видеокартах рост производительности будет продолжаться.

Таким образом, использование технологии CUDA оправданно и позволяет добиться сокращения времени работы программы, что является важным фактором для задач биоинформатики, где необходимо обрабатывать большие массивы данных.

6. Семплирование иерархий транскрипционных факторов. Для того чтобы проанализировать взаимосвязь геномных профилей и факторов транскрипции, определенных экспериментальным путем, воспользуемся существующей биологической классификацией.

Вообще говоря, транскрипционные факторы можно классифицировать несколькими способами:

- по механизму действия
- по белковой структуре;
- по происхождению.

По механизму действия транскрипционные факторы разбиваются на три класса:

1. **Главные факторы транскрипции**, вовлеченные в образование инициационного комплекса. Они присутствуют во всех клетках и взаимодействуют с кор-промотором генов, транскрибируемых РНК-полимеразой.

2. Факторы, взаимодействующие с upstream-участками ДНК – областями, расположенными до промотора, лежащими относительно него с другой стороны от кодирующей области гена.

3. Индуцируемые факторы — сходны с факторами, взаимодействующими с upstream-участками ДНК, но требуют активации либо ингибирования.

- По регуляторной функции.

1. Конститутивные — присутствуют всегда во всех клетках – главные факторы транскрипции.

2. Активируемые — активны в определенных условиях.

2.1. Участвующие в развитии организма (клеточно-специфичные) – факторы, экспрессия которых строго контролируется, но, начав экспрессироваться, не требуют дополнительной активации.

2.2. Сигнал-зависимые — требующие внешнего сигнала для активации**.**

2.2.1. Внеклеточные сигнал-зависимые – ядерные рецепторы.

2.2.2. Внутриклеточные сигнал-зависимые — активируются низкомолекулярными внутриклеточными соединениями.

2.2.3. Мембраносвязанные рецептор-зависимые — фосфорилируются киназами сигнального каскада.

2.2.3.1. Резидентные ядерные факторы – находятся в ядре независимо от активации.

2.2.3.2. Латентные цитоплазматические факторы – в неактивном состоянии локализованы в цитоплазме, после активации транспортируются в ядро.

- Структурная классификация.

По данному признаку факторы транскрипции классифицируют на основании сходства первичной структуры (что предполагает и сходство третичной структуры) ДНК-связывающих доменов [13,14]. Данная классификация представлена деревом, узлы которого соответствуют некоторой общей структуре ДНК-связывающих доменов, присущей транскрипционным факторам, соответствующие весовые матрицы которых содержатся в данном узле.

Для того чтобы оценить актуальность и применимость введенной нами конструкции геномных профилей, будем использовать структурную классификацию транскрипционных факторов, так как она является наиболее доступной и хранит данные в структурном виде в виде дерева. Проведем семплирование исходной классификации и проанализируем, насколько коррелирует семплированная выборка с полученными геномными профилями. Под семплированием будем понимать процесс получения случайного дерева классификации, по структуре похожего на существующую биологическую классификацию. Для получения семплов будем пользоваться двумя операциями:

1) Обмен значениями — два узла дерева меняются набором весовых матриц, соответствующих данной структуре. Пример операции изображен на рис. 16.

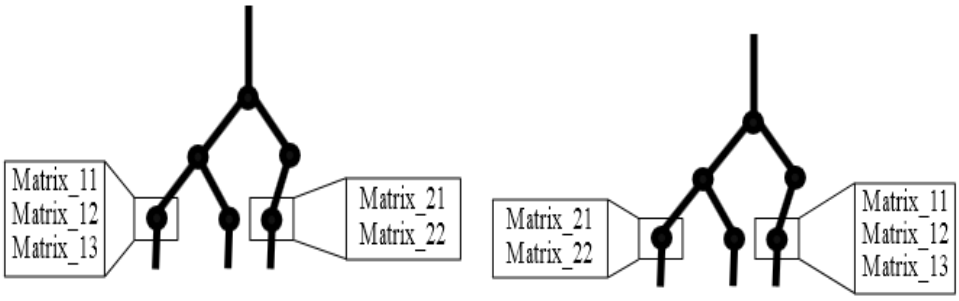


Рис.16. Обмен значениями двух узлов до (слева) и после (справа) операции

2) Обмен дочерней классификацией – два узла меняются местами вместе с оставшейся под ними дочерней классификацией. Пример операции изображен на рис. 17.

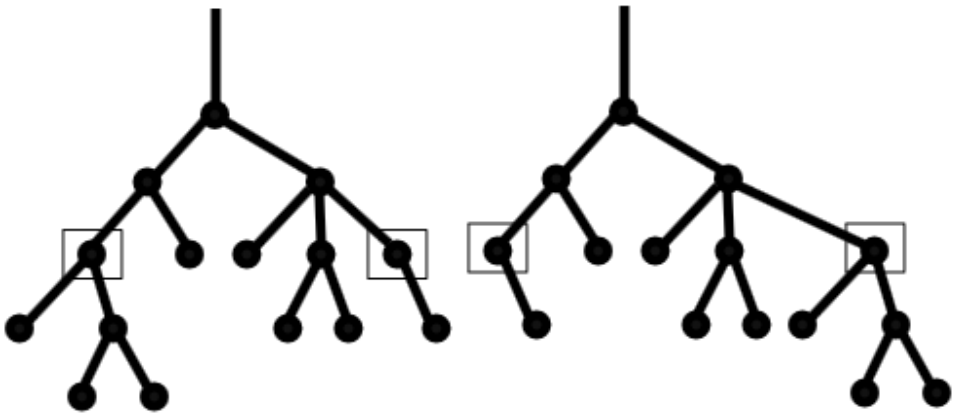


Рис.17. Обмен дочерней классификацией до (слева) и после (справа) операции

Таким образом, применив на каждом уровне заданное количество операций обмена, получим семплированное дерево классификации. Подробнее алгоритм изображен на рис. 18.

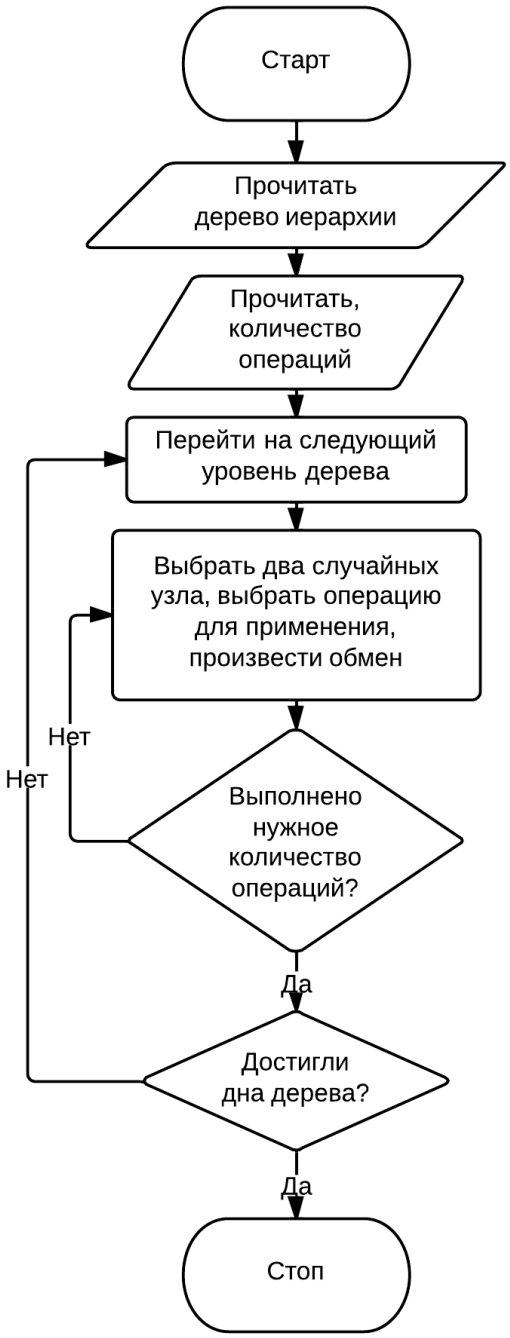


Рис.18. Алгоритм семплирования дерева иерархии

7. Сравнение матрицы расстояний и иерархии. Применив алгоритм семплирования, получим выборку T, T_1, T_2, …, T_N, где T — оригинальное дерево иерархии факторов транскрипции, а T_i — дерево, полученное в результате семплирования. Для каждого дерева построим матрицу расстояний, где номер строки/столбца есть номер весовой матрицы в списке всех весовых матриц для факторов транскрипции, а значения элементов матрицы расстояний суть число рёбер в минимальном пути дерева между вершинами, содержащими выбранные весовые матрицы (так как одна весовая матрица может содержаться в нескольких вершинах классификации). Получим набор матриц расстояний D, D_1, D_2, …, D_N. Для анализа

полученных данных построим матрицу расстояний для геномных профилей. Т.к. геномные профили — это вектора длины 20, мы можем вычислить расстояние между парами профилей с помощью евклидовой метрики. Следовательно, получим матрицу P, элементами которой являются расстояния между соответствующими парами геномных профилей. Для сравнения полученных результатов вычислим линейный коэффициент корреляции между матрицей расстояний геномных профилей и каждой из матриц для деревьев классификации. Коэффициент корреляции рассчитывается по формуле (суммирование проводится по всем элементам матриц):

$$r(X,Y) = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2}\sqrt{\sum(y_i - \bar{y})^2}},$$

где $\bar{x} = \frac{1}{n}\sum x_i$ и $\bar{y} = \frac{1}{n}\sum y_i$ – средние значения элементов матриц.

Таким образом, мы получили набор r, r_1, r_2, …, r_N значений корреляции, где r = r(P, T), r_i = r(P, T_i). На основании полученных данных мы можем построить гистограмму распределения коэффициентов корреляции случайных деревьев и, проанализировав разницу корреляции случайных деревьев и существующей иерархии, сделать вывод о биологической применимости геномных профилей для факторов транскрипции.

8. Результаты. С помощью созданной авторами программы GenomeSignal были построены геномные профили для весовых матриц библиотеки TRANSFAC. При этом реализация алгоритма поиска потенциальных сайтов связывания для GPU показала **семидесятикратный** прирост производительности на видеокарте GeForce GTX 650 по сравнению с последовательными вычислениями.

Для того чтобы проанализировать взаимосвязь геномных профилей и факторов транскрипции, была сгенерирована выборка из N=400 случайных деревьев, основанных на существующем дереве классификации. На каждом уровне исходного дерева проводилось C=5000 операций обмена, вариант обмена выбирался каждый раз случайным образом, так же случайно выбиралась пара узлов для обмена. Таким образом, мы построили 400 случайных деревьев и вычислили соответствующие им матрицы расстояний. Вычислив корреляцию полученных данных с матрицей расстояний геномных профилей, мы получили следующие результаты.

Для того чтобы получить информацию о распределении коэффициентов корреляции, была построена соответствующая гистограмма. Для этого отрезок между минимальным и

максимальным значениями корреляции [-0,00211819; 0,0700183] был разбит на N=20 полуинтервалов одной длины, и был произведен подсчет количества значений, попавших в тот или иной полуинтервал.

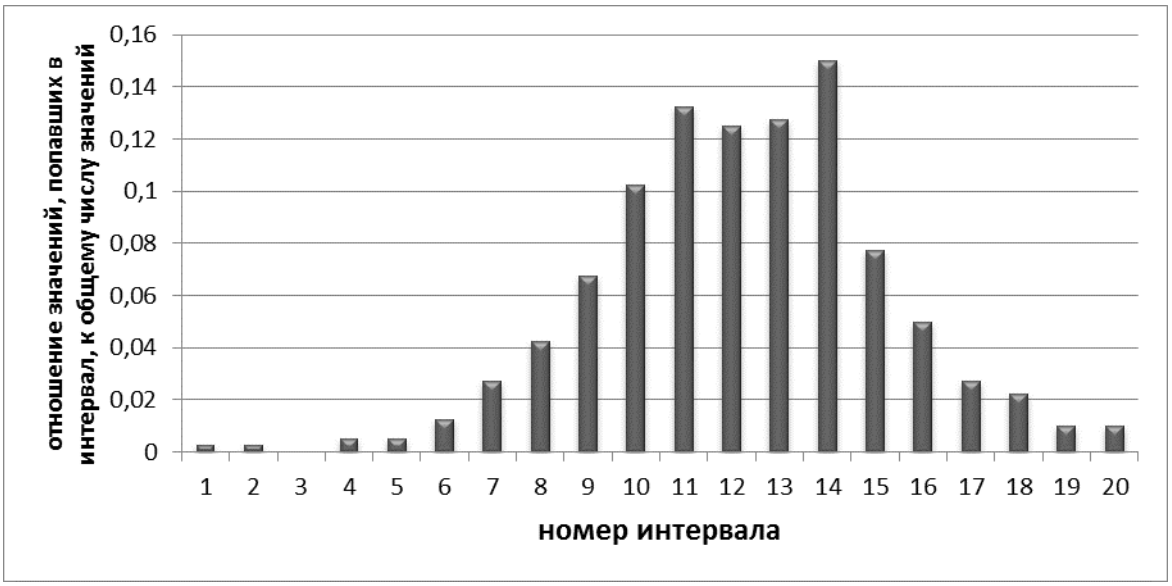


Рис.19. Гистограмма распределения коэффициентов корреляции

Также был построен график значений коэффициентов корреляции.

Рис.20. График полученных значений корреляции для случайно сгенерированных деревьев

Можно видеть, что корреляция с существующими данными в среднем в два раза выше, чем со случайно созданной иерархией транскрипционных факторов. На основе анализа полученных данных был сделан вывод о том, что получение и изучение свойств геномных профилей является оправданным и хорошо соотносится с существующими данными, полученными экспериментальным путем в лабораториях.

9. Заключение. Разработанная программа позволяет строить распределение сайтов на хромосоме, их профили, а также графически отображать результаты в виде гистограмм. Данная программа написана с целью быстрого построения геномных профилей на последовательностях ДНК, необходимых для анализа и сравнения структур весовых матриц. Реализованы несколько алгоритмов кластеризации геномных профилей. Реализованы алгоритмы построения матриц расстояний с помощью новой введенной конструкции «геномные профили весовых матриц» и с помощью существующей иерархии транскрипционных факторов. Реализован алгоритм получения случайных деревьев иерархии транскрипционных факторов, основанный на многократном изменении существующего дерева классификации. Проведен анализ и сделан вывод о целесообразности использования геномных профилей при исследовании влияния сайтов связывания, находящихся в окрестности некоторого данного сайта.

Параллельная реализация алгоритма поиска потенциальных сайтов связывания с транскрипционными факторами позволяет значительно ускорить работу программы на современных видеоадаптерах, поддерживающих технологию CUDA. Эксперименты показали семидесятикратный рост производительности по сравнению с последовательными вычислениями на центральном процессоре. Использование программы на новейших мощных видеоадаптерах позволит получить более высокие показатели скорости работы.

Список литературы

1. Колчанов Н.А., Гончаров С.С., Лихошвай В.А., Иванисенко В.А. Системная компьютерная биология // СО РАН, 2008. С. 10–70.
2. Peter J. Park ChIP–seq: advantages and challenges of a maturing technology // Nature Reviews Genetics. 2009. Vol. 10. P. 669-680.
3. Жимулёв И.Ф. Общая и молекулярная генетика // Новосибирск: Изд-во НГУ, 2002. 458 с.
4. Kozak, M. Initiation of translation in prokaryotes and eukaryotes // Gene. 1999. Vol. 234(2). P. 187–208.
5. Veenstra, G.J., A.P. Wolffe. Gene-selective developmental roles of general transcription factors // Trendsin Biochemical Sciences. 2001. Vol. 26(11). P. 665–671.

6. TRANSFAC: [Электронный ресурс] // Biobase. URL: http://www.gene-regulation.com/pub/databases.html. (Дата обращения: 01.05.2013).
7. Harr, R., M. Häggström, and P. Gustafsson. Search algorithm for pattern match analysis of nucleic acid sequences // Nucleic Acids Research. 1983. Vol. 11(9). P. 2943–2957.
8. Stormo, G.D., et al., Use of the 'Perceptron' algorithm to distinguish translational initiation sites in E. coli // Nucleic Acids Research. 1982. Vol. 10(9). P. 2997–3011.
9. Bulyk, M.L., P.L. Johnson, G.M. Church. Nucleotides of transcription factor binding sites exert interdependent effects on the binding affinities of transcription factors // Nucleic Acids Res, 2002. Vol. 30(5). P. 1255–61.
10. Zhou, Q., J.S. Liu. Modeling within-motif dependence for transcription factor binding site predictions // Bioinformatics. 2004. Vol. 20(6). P. 16–909.
11. Jen-Hsun Huang on NVidia GTC 2013 // NVidia GTC 2013. Материалы конференции.
12. CUDA C Programming Guide: [Электронный ресурс] // NVidia. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide. (Дата обращения: 01.05.2013).
13. Matys V, et.al. TRANSFAC and its module TRANSCompel: transcriptional gene regulation in eukaryotes // Nucleic Acids Res. 2006. Vol. 34.
14. Stegmaier P, Kel AE, Wingender E. Systematic DNA-binding domain classification of transcription factors // Genome informatics. International Conference on Genome Informatics. 2004. Vol. 15 (2). P. 276–86.

UDK: 519.688

Title: Program for building and clustering of genomic profiles using GPU

Authors:

Nikitin Sergey Ilyich (Novosibisrk State University),

Cheryomushkin Evgeny Sergeevich (Ershov Institute of Informatics Systems)

Abstract: Before RNA transcription starts, special segments of DNA form a complex of regulatory proteins called transcription factors. This complex allows RNA polymerase to be bound to DNA and to start reading RNA. It is a difficult problem to search binding sites on DNA because of many factors influencing the binding. In particular, other sites in the vicinity of a given site may influence binding. To reveal those dependencies the authors introduce histograms of density distribution of binding sites, called genomic profiles.

The software package developed in the scope of this work allows building genomic profiles using the prediction of binding sites by a weight matrices algorithm for different implementations: for multicore CPU’s or NVidia GPU’s using CUDA. In addition, the software allows clustering

genomic profiles using K-means clustering and hierarchical clustering. The algorithm allows to build samples – random transcription factors hierarchies based on the existing experimental structural classification to estimate the correspondence between genomic profiles construction and the existing classification. The analysis of correspondence of genomic profiles with biological classification of transcription factors was developed using the sampling algorithm.

Keywords: DNA regulation, Genome, Position weight matrices, Transcription factors, transcription factor binding sites, Clustering, Nvidia CUDA, GPU computing

UDK: 519.681.3, 519.681.2

Title: Causality versus True Concurrency in the Setting of Real-Time Models

Author(s):

Nataliya S. Gribovskaya (A.P. Ershov Institute of Informatics Systems, SB RAS)

Abstract: The contribution of the paper is to clarify connections between real-time models of concurrency. In particular, we defined a category of timed causal trees and investigated how it relates to other categories of timed models. Moreover, using a larger model called timed event trees, we constructed an adjunction from the category of timed causal trees to the category of timed event structures. Thereby we showed that timed causal trees are more trivial than timed event structures because they reflect only one aspect of true concurrency, causality, and they apply causality without a notion of event. On the other hand, the first model is more expressive than the latter in that possible runs of a timed causal tree can be defined in terms of a tree without restrictions, but the set of the possible runs of any event structure must be closed under the shuffling of concurrent transitions.

Keywords: real-time models, true concurrency, causality, relations, unification, category theory

1. Introduction. In recent decades, category theoretical approaches have been actively used for the specification and investigation of concurrent systems and processes. We will mention just one example, which is directly related to the concurrency theory. The category theory has helped us to classify and unify various models for concurrency and has provided an abstract language for expressing relationships between seemingly very different models. The basic goal is to formulate the fact that one model is more expressive than another in terms of an 'embedding' or coreflection (reflection) — the category theoretical notion defined as an adjunction, in which the unit (counit) is an isomorphism. In the setting of this approach, models are represented as categories: each model is equipped with a notion of morphism that shows how one model instance can be simulated by another. Moreover, the existence of (co)reflection between models allows us to translate concepts and properties from one model to another.

At present, the concurrency theory has a great variety of formal models that can be classified based on different principles. For example, concurrent models are split to interleaving models and true concurrent models. For interleaving models, such as synchronization trees, causal trees and transition systems, the concurrency is simulated by a sequence of actions. For true concurrent models, such as event structures, transition systems with independence, labelled asynchronous transition systems, causal trees and Petri nets, the concurrency is modelled implicitly through the relation of independence.

In [3, 5] Winskel, Nielsen and Joyal have applied the category theory to unify the many models for concurrency and to establish the relationships between them. They have shown that

the categories of such models as synchronization trees, transition systems, event structures, transition systems with independence and asynchronous transition systems are related by coreflections. In particular, they have found out the following facts. Intuitively, synchronization trees are transition systems with no cyclic behaviour. Moreover, a synchronization tree may be transferred to a special kind of an event structure with an empty independence relation. The transition systems may be regarded as transition systems with independence in which the independence relation is empty. An event structure may be translated to a special type of a transition system with independence. Finally, transition systems with independence may be considered as asynchronous transition systems, which have at most one transition with the same label between two same states. Later Nielsen and Winskel proved that there exists a coreflection between Petri nets and asynchronous transition systems (see [4]). In [2] Fröschle and Lasota integrated a new model, the causal trees of Darondeau and Degano, into Winskel and Nielsen's framework. Also they have shown that there is an adjunction from causal trees to event structures. Causal trees are some variant of synchronization trees with enriched action labels that supply information about which transitions causally depend on each other. Thereby, they reflect the only one aspect of true concurrency, causality. On the other hand, there is one aspect in which event structures are less expressive than causal trees: their notion of run is induced abstractly by the consistency and causal dependency relation. In particular, this means the set of runs of any event structure is closed under the shuffling of concurrent transitions.

More recently, great efforts have been made to develop formal methods for real-time systems. These are systems whose correctness depends crucially upon real-time considerations. As a result, time extensions of concurrent models such as timed automata, times synchronization trees, timed transition systems, timed event structures, and timed Petri nets have appeared and have been investigated. However, only a few examples of the category theoretical classification for timed models are described in literature.

The contribution of the paper is to show the applicability of the general categorical framework proposed by Winskel and Nielsen and to clarify connections between real-time models of concurrency. In particular, we defined categories for such models as timed transition systems, timed synchronization trees, timed causal trees and timed event structure, and investigated how they relate with each other. Moreover, using a larger model called timed event trees we showed the existence of an adjunction from the category of timed causal trees to the category of timed event structures.

The rest of the paper is organized as follows. The basic notions and notations of the category theory are introduced in Section 2. In the next section, we define categories for timed extensions of concurrent models and establish some of their properties. Five subsections of Section 3 describe five different models: timed transition systems, timed synchronization trees, timed

causal trees, timed event structures and timed event trees. Relations between timed models for concurrency are introduced in Section 4, which consists of five subsections. In the first subsection a coreflection between the category of timed causal trees and the category of timed synchronization trees is exhibited. The existence of a coreflection between the category of timed causal trees and the category of timed event trees is shown in Subsection 4.2. The next subsection proves the existence of a reflection between the category of timed event structures and the category of timed event trees. In the fourth subsection, the construction of a coreflection between the category of timed event structures and the category of timed synchronization trees is described. Each of above subsections consists of definitions of two functors between two certain categories, some useful propositions and the main theorem, which asserts the existence of (co)reflection between the categories. Subsection 4.5 recapitulates the obtained results. Section 5 is the conclusion of the paper.

2. Basics of the Category Theory. In this section we will briefly recall some basic notions and notations from the category theory [1]. Let us start with the definition of a category.

Definition 1. A category $\mathcal{C}$ *consists of the following:*

- *a class* $|\mathcal{C}|$*, whose elements will be called "objects of the category";*
- *for every pair* A*,* B *of objects, a set* $\mathcal{C}(A,B)$*, whose elements will be called "morphisms" or "arrows" from* A *to* B*;*
- *for every triple* A*,* B*,* C *of objects, a composition law* $\mathcal{C}(A,B)\times\mathcal{C}(B,C)\longrightarrow\mathcal{C}(A,C)$*. The composite of the pair* (f,g) *will be written* $g\circ f$ *or just* gf*;*
- *for every object* A*, a morphism* $I_A\in\mathcal{C}(A,A)$*, called the identity on* A*.*

These data are subject to the following axioms.

- Associativity axiom*: given morphisms* $f\in\mathcal{C}(A,B)$*,* $g\in\mathcal{C}(B,C)$*,* $h\in\mathcal{C}(C,D)$ *the following equality holds:* $h\circ(g\circ f)=(h\circ g)\circ f$*;*
- Identity axiom*: given morphisms* $f\in\mathcal{C}(A,B)$*,* $g\in\mathcal{C}(B,C)$*, the following equalities hold:* $1_B\circ f=f$*,* $g\circ 1_B=g$*.*

Now we adduce the notion of a functor (or a "homomorphism of categories") with some of their properties.

Definition 2. A functor F *from a category* $\mathcal{C}$ *to a category* $\mathcal{D}$ *consists of the following:*

- *a mapping* $|\mathcal{C}|\longrightarrow|\mathcal{D}|$ *between the classes of objects of* $\mathcal{C}$ *and* $\mathcal{D}$*; the image of* $A\in\mathcal{C}$ *is written* $F(A)$ *or just* FA*;*

- *for every pair of objects A, A' of $\mathcal{C}$, a mapping $\mathcal{C}(A, A') \longrightarrow \mathcal{D}(FA, FA')$; the image of $f \in \mathcal{C}(A, A')$ is written $F(f)$ or just Ff.*

These data are subject to the following axioms:

- *for every pair of morphisms $f \in \mathcal{C}(A, A')$, $g \in \mathcal{C}(A', A'')$ $F(g \circ f) = F(g) \circ F(f)$;*

- *for every object $A \in \mathcal{C}$ $F(1_A) = 1_{FA}$.*

Definition 3. *Consider a functor $F : \mathcal{C} \to \mathcal{D}$ and for every pair of objects $A, A' \in \mathcal{C}$, the mapping $\mathcal{C}(A, A') \to \mathcal{D}(FA, FA')$, $f \mapsto Ff$.*

- *The functor F is faithful when the above mentioned mappings are injective for all A, A';*
- *The functor F is full when the above mentioned mappings are surjective for all A, A';*
- *The functor F is full and faithful when the above mentioned mappings are bijective for all A, A';*
- *The functor F is an isomorphism of categories when it is full and faithful and induces a bijection $|\mathcal{C}| \longrightarrow |\mathcal{D}|$ on the classes of objects.*

There is a notion of natural transformations in the category theory, which is an adaptation of the notion of a "homotopy" between two continuous functions from one space to another.

Definition 4. *Consider two functors $F, G : \mathcal{C} \to \mathcal{D}$ from a category $\mathcal{C}$ to a category $\mathcal{D}$.* A natural transformation *$\alpha : F \Rightarrow G$ from F to G is a class of morphisms $(\alpha_A : FA \to GA)_{A \in \mathcal{C}}$ of $\mathcal{D}$ indexed by the objects of $\mathcal{C}$ and such that for every morphism $f : A \to A'$ in $\mathcal{C}$, $\alpha_{A'} \circ F(f) = G(f) \circ \alpha_A$.*

One of the basic conceptions of the category theory is a notion of adjoint functors. There are various definitions for adjoint functors. Their equivalence is elementary but not at all trivial. We will use the definitions via reflections and coreflections along functors.

Definition 5. *Let $F : \mathcal{C} \to \mathcal{D}$ be a functor and B an object of $\mathcal{D}$.* A reflection of B along F *is a pair (R_B, η_B) where R_B is an object of $\mathcal{C}$, $\eta_B : B \to F(R_B)$ is a morphism of $\mathcal{D}$, and if $A \in |\mathcal{C}|$ is an object of $\mathcal{C}$ and $b : B \to F(A)$ is a morphism of $\mathcal{D}$, then there exists a unique morphism $a : R_B \to A$ in $\mathcal{C}$ such that $F(a) \circ \eta_B = b$.*

Definition 6. *Let $F : \mathcal{C} \to \mathcal{D}$ be a functor and B an object of $\mathcal{D}$.* A coreflection of B along F *is a pair (R_B, ϵ_B) where R_B is an object of $\mathcal{C}$, $\epsilon_B : F(R_B) \to B$ is a morphism of $\mathcal{D}$, and if $A \in |\mathcal{C}|$ is an object of $\mathcal{C}$ and $b : F(A) \to B$ is a morphism of $\mathcal{D}$, then there exists a unique morphism $a : A \to R_B$ in $\mathcal{C}$ such that $\epsilon_B \circ F(a) = b$.*

Definition 7. *A functor $R : \mathcal{D} \to \mathcal{C}$ is* left adjoint *to the functor $F : \mathcal{C} \to \mathcal{D}$ (and F is* right adjoint *to R) when there exists a natural transformation $\eta : 1_{\mathcal{D}} \Rightarrow F \circ R$, called the unit of the adjunction, such that for every $B \in \mathcal{D}$, a pair (RB, η_B) is a reflection of B along F.*

Definition 8. *A functor $R : \mathcal{D} \to \mathcal{C}$ is* right adjoint *to the functor $F : \mathcal{C} \to \mathcal{D}$ (and F is* left adjoint *to R) when there exists a natural transformation $\epsilon : F \circ R \Rightarrow 1_{\mathcal{D}}$, called the counit of the adjunction, such that for every $B \in \mathcal{D}$, a pair (RB, ϵ_B) is a coreflection of B along F.*

We will call an adjunction in which the unit (the counit) is a natural isomorphism as *a coreflection* (*a reflection*).

3. Models for Concurrency. In this section we study the timed extensions of five different concurrent models. Four of them are well-known interleaving and true concurrency models, and the fifth one is called event trees and embeds causal trees as well as event structures. Event trees are like event structures because causality and concurrency are event-based, global notions. They are like causal trees because their possible runs are specified explicitly by a tree.

We start by introducing of timed variants of the models, and then we define categories for them.

3.1. Timed Transition Systems. Let $\mathbf{R}$ be a set of non-negative reals and L be a finite alphabet of actions. Consider the definition of timed transition systems.

Definition 9. *A* timed transition system $\mathcal{T}$ over an alphabet L *is a tuple (S, s_{in}, L, T), where S is a set of states and s_{in} is the initial state, $T \subseteq S \times L \times \mathbf{R} \times \mathbf{R} \times S$ is a set of transitions such that for all $(s, \sigma, eot, lot, s') \in T$ we have $eot \le lot$. We will write $s \xrightarrow[eot,\ lot]{\sigma} s'$ to denote a transition $(s, \sigma, eot, lot, s')$.*

Let us define the behaviour of timed transition systems.

Definition 10. *Let $\mathcal{T}$ be a timed transition system over L.*

A configuration *of $\mathcal{T}$ is a pair $\langle s, \nu \rangle$, where s is a state and ν is a current global time moment.*

A run *of $\mathcal{T}$ is a sequence $\gamma = \langle s_0, \nu_0 \rangle \overset{\sigma_1}{\to} \langle s_1, \nu_1 \rangle \ \ldots \ \langle s_{n-1}, \nu_{n-1} \rangle \overset{\sigma_n}{\to} \langle s_n, \nu_n \rangle$ such that $\nu_1 \le \ldots \le \nu_n$ and for all $1 \le i \le n$ there is a transition $s_{i-1} \xrightarrow[eot_i,\ lot_i]{\sigma_i} s_i$ such that $eot_i \le \nu_i \le lot_i$. Here, $s_0 = s_{in}$ and ν_0 is defined to be 0.*

We are now ready to introduce the category of timed transition systems.

Definition 11. *Given timed transition systems $\mathcal{T} = (S,\ s_{in},\ L,\ T)$ and $\mathcal{T}' = (S',\ s'_{in},\ L',\ T')$, a pair (μ, λ) is a* morphism *between $\mathcal{T}$ and $\mathcal{T}'$, if $\mu : S \to S'$ and $\lambda : L \to L'$ are functions such that $\mu(s_{in}) = s'_{in}$, and if $(s, \sigma, eot, lot, s') \in T$, then $(\mu(s), \lambda(\sigma), eot', lot', \mu(s')) \in T'$ for some real numbers eot' and lot' such that $eot' \le eot$ and $lot \le lot'$.*

Timed transition systems and morphisms between them form a category of timed transition systems, **TTS**, in which the composition of two morphisms $(\mu, \lambda) : \mathcal{T} \to \mathcal{T}'$ and $(\mu', \lambda') : \mathcal{T}' \to \mathcal{T}''$ is defined as $(\mu', \lambda') \circ (\mu, \lambda) \coloneqq (\mu' \circ \mu, \lambda' \circ \lambda)$, and the identity morphism is a pair of the identity functions.

Lemma 1. *Given a morphism $(\mu, \lambda) : \mathcal{T} \to \mathcal{T}'$ of* **TTS**, *if $\langle s_0, \nu_0 \rangle \overset{\sigma_1}{\to} \langle s_1, \nu_1 \rangle \ \ldots \ \langle s_{n-1}, \nu_{n-1} \rangle \overset{\sigma_n}{\to} \langle s_n, \nu_n \rangle$ is a run of $\mathcal{T}$ then $\langle \mu(s_0), \nu_0 \rangle \overset{\lambda(\sigma_1)}{\to} \langle \mu(s_1), \nu_1 \rangle \ \ldots \ \langle \mu(s_{n-1}), \nu_{n-1} \rangle \overset{\lambda(\sigma_n)}{\to} \langle \mu(s_n), \nu_n \rangle$ will be a run of $\mathcal{T}'$.*

3.2. Timed Synchronization Trees. Now we contemplate the definition of timed synchronization trees.

Definition 12. *A* timed synchronization tree $\mathcal{S}$ *is a timed transition system (S, s_{in}, L, T) such that*

(i) for all $s \in S$ there exists a sequence $s_{in} \xrightarrow[eot_1,\ lot_1]{\sigma_1} s_1 \ \ldots \ s_{k-1} \xrightarrow[eot_k,\ lot_k]{\sigma_k} s_k$ $(k \geq 0)$ such that $s = s_k$;

(ii) for all sequence $s_0 \xrightarrow[eot_1,\ lot_1]{\sigma_1} s_1 \ \ldots \ s_{k-1} \xrightarrow[eot_k,\ lot_k]{\sigma_k} s_k$ $(k \geq 0)$ it holds if $s_0 = s_k$ then $k = 0$;

(iii) if $s' \xrightarrow[eot,\ lot]{\sigma} s$ and $s'' \xrightarrow[eot',\ lot']{\sigma'} s$, then $s' = s''$, $\sigma = \sigma'$, $eot = eot'$ and $lot = lot'$.

Write **TST** for the full subcategory of timed synchronization trees in **TTS**.

3.3. Timed Causal Trees. In this subsection we introduce the timed extension of causal trees, which are a generalization of synchronization trees.

Definition 13. *A* timed causal tree $\mathcal{C}$ *is a tuple $(S, s_{in}, L, T, <)$ where $(S,\ s_{in},\ L,\ T)$ is a timed synchronization tree and $< \subseteq T \times T$, the* causal dependency relation*, is a strict order such that for all transitions $(s,\ \sigma,\ eot,\ lot,\ s')$ and $(s'',\ \sigma',\ eot',\ lot',\ s''')$ of $\mathcal{C}$ if $(s, \sigma, eot, lot, s') < (s'', \sigma', eot', lot', s''')$, then there exists a sequence $s' \xrightarrow[eot_1,\ lot_1]{\sigma_1} s_1 \ \ldots \ s_{k-1} \xrightarrow[eot_k,\ lot_k]{\sigma_k} s''$ for some $k \geq 0$.*

We will say that two transitions $(s, \sigma, eot, lot, s')$ and $(s'', \sigma', eot', lot', s''')$ of $\mathcal{C}$ are *consistent* (denoted $(s, \sigma, eot, lot, s')\ Con\ (s'', \sigma', eot', lot', s''')$) iff either $(s,\ \sigma,\ eot,\ lot,\ s') = (s'',\ \sigma',\ eot',\ lot',\ s''')$ or there exists a sequence $s_0 \xrightarrow[eot_1,\ lot_1]{\sigma_1} s_1 \ \ldots \ s_{k-1} \xrightarrow[eot_k,\ lot_k]{\sigma_k} s_k$ $(k \geq 0)$ such that $(s' = s_0 \ \wedge \ s'' = s_k)$ or $(s''' = s_0 \ \wedge \ s = s_k)$. A *run* of $\mathcal{C} = (S, s_{in}, L, T, <)$ is a sequence $\gamma = \langle s_0, \nu_0 \rangle \overset{\sigma_1}{\underset{K_1}{\to}} \langle s_1, \nu_1 \rangle \ \ldots \ \langle s_{n-1}, \nu_{n-1} \rangle \overset{\sigma_n}{\underset{K_n}{\to}} \langle s_n, \nu_n \rangle$ such that $\langle s_0, \nu_0 \rangle \overset{\sigma_1}{\to} \langle s_1, \nu_1 \rangle \ \ldots \ \langle s_{n-1}, \nu_{n-1} \rangle \overset{\sigma_n}{\to} \langle s_n, \nu_n \rangle$ is a run of (S, s_{in}, L, T) and $K_i = \{j \mid 1 \leq j \leq i,\ (s_{j-1}, \sigma_j, eot_j, lot_j, s_j) < (s_{i-1}, \sigma_i, eot_i, lot_i, s_i)\}$ for all $1 \leq i \leq n$.

We are ready to equip timed causal trees with a notion of morphism and thus define a category of timed causal trees.

Definition 14. *Given timed causal trees* $\mathcal{C} = (S,\ s_0,\ L,\ T, <)$ *and* $\mathcal{C}' = (S',\ s'_0,\ L',\ T',\ <')$*, a pair* (μ, λ) *is a* morphism *between* $\mathcal{C}$ *and* $\mathcal{C}'$*, if* (μ, λ) *is a morphism between timed synchronization trees* $(S,\ s_0,\ L,\ T)$ *and* $(S',\ s'_0,\ L',\ T')$ *and for all transitions* $(s, \sigma, eot, lot, s_1)$ *and* $(s_2, \sigma_1, eot_1, lot_1, s_3)$ *of* $\mathcal{C}$*, if* $(s, \sigma, eot, lot, s_1)$ *Con* $(s_2, \sigma_1, eot_1, lot_1, s_3)$ *and* $(\mu(s),\ \lambda(\sigma),\ eot',\ lot',\ \mu(s_1)) <' (\mu(s_2),\ \lambda(\sigma_1),\ eot'_1,\ lot'_1,\ \mu(s_3))$ *for some* $eot',\ lot',\ eot'_1,\ lot'_1 \in \mathbf{R}$ *such that* $eot' \le eot$, $lot \le lot'$, $eot'_1 \le eot_1$ *and* $lot_1 \le lot'_1$*, then* $(s, \sigma, eot, lot, s_1) < (s_2, \sigma_1, eot_1, lot_1, s_3)$.

Timed causal trees and their morphisms form a category of timed causal trees, **TCT**.

Lemma 2. *Given a morphism* $(\mu, \lambda) : \mathcal{C} \to \mathcal{C}'$ *of* **TCT**, *if* $\gamma = \langle s_0, \nu_0\rangle \overset{\sigma_1}{\underset{K_1}{\to}} \langle s_1, \nu_1\rangle \ \dots\ \langle s_{n-1}, \nu_{n-1}\rangle \overset{\sigma_n}{\underset{K_n}{\to}} \langle s_n, \nu_n\rangle$ *is a run of* $\mathcal{C}$*, then* $\gamma' = \langle \mu(s_0), \nu_0\rangle \overset{\lambda(\sigma_1)}{\underset{K'_1}{\to}} \langle \mu(s_1), \nu_1\rangle \ \dots\ \langle \mu(s_{n-1}), \nu_{n-1}\rangle \overset{\lambda(\sigma_n)}{\underset{K'_n}{\to}} \langle \mu(s_n), \nu_n\rangle$ *will be a run of* $\mathcal{C}'$ *for some* $K'_1, \dots, K'_n$ *such that* $K'_i \subseteq K_i$ *for all* $1 \le i \le n$.

3.4. Timed Event Structures. This subsection is dedicated to the most popular true concurrency model — timed event structures. Let us first give the definition of this model.

Definition 15. *A* timed event structure *is a tuple* $\mathcal{E} = (E, <, Con, L, l, Eot, Lot)$*, where* E *is a set of events;* $< \subseteq E \times E$ *is a strict order (the* causality relation*), satisfying the* principle of finite causes*:* $\forall e \in E \circ e\downarrow = \{e' \in E \mid e' < e\}$ *is finite;* $Con \subseteq 2^E$ *(the* consistency relation*) consists of finite subsets of events which can occur together in a run, satisfying the following* principles*:* $\forall e \in E \circ \{e\} \in Con$; $Y \subseteq X \in Con \Rightarrow Y \in Con$ *and* $X \in Con \wedge e < e' \in X \ \Rightarrow X \cup \{e\} \in Con$; L *is a set of actions;* $l : E \to L$ *is a labelling function and* $Eot,\ Lot : E \to \mathbf{R}$ *are functions of the earliest and the latest occurrence times of events, satisfying the following:* $Eot(e) \le Lot(e)$ *for all* $e \in E$.

Let $C \subseteq E$. Then C is *left-closed* iff $\forall e, e' \in E \circ e \in C \ \wedge \ e' < e \Rightarrow e' \in C$; C is *consistent* iff $C \in Con$; C is a *configuration of* $\mathcal{E}$ iff C is left-closed and consistent. Let $\mathbf{C}(\mathcal{E})$ denote the set of all finite configurations of $\mathcal{E}$.

An execution of a timed event structure is a *timed configuration* which consists of a configuration and a timing function recording global time moments at which events occur and satisfies some additional requirements. Let $\mathcal{E} = (E, <, Con, L, l, Eot, Lot)$ be a timed event structure, $C \in \mathbf{C}(\mathcal{E})$, and $T : C \to \mathbf{R}$. Then $TC = (C, T)$ is a *timed configuration* of $\mathcal{E}$ iff $\forall\ e \in C \circ\ Eot(e) \le T(e) \le Lot(e)$ and $\forall\ e, e' \in C \circ\ e < e' \ \Rightarrow\ T(e) \le T(e')$. Informally speaking, the first condition expresses that an event can occur at a time when its timing constraints are met; and the second condition states that for any two events e and e' occurred if e causally precedes e', then e should temporally precede e'. We use $\mathbf{TC}(\mathcal{E})$ to denote the set of timed configurations of $\mathcal{E}$.

Let $\mathcal{E}$ be a timed event structure and $TC = (C,T), TC' = (C',T') \in \mathbf{TC}(\mathcal{E})$. We will write $TC \xrightarrow[d]{e} TC'$ iff $C \cup \{e\} = C'$, and $T'|_C = T$ and $T'(e) = d$. A *run* of $\mathcal{E}$ is a sequence of the form $TC_0 \xrightarrow[d_1]{e_1} TC_1 \xrightarrow[d_2]{e_2} \dots \xrightarrow[d_n]{e_n} TC_n$, where $n \geq 0$ and $TC_0 = (\varnothing, \varnothing)$ is the initial timed configuration.

Now let us recall the notion of morphism between timed event structures.

Definition 16. *Let $\mathcal{E} = (E, <, Con, L, l, Eot, Lot)$ and $\mathcal{E}' = (E', <', Con', L', l', Eot', Lot')$ be timed event structures. A pair (μ, λ), where $\mu : E \to E'$ and $\lambda : L \to L'$ are functions, is called a* morphism*, if $l' \circ \mu = \lambda \circ l$ and for all $C \in \mathbf{C}(\mathcal{E})$ the following holds:*

- $\mu\ C \in \mathbf{C}(\mathcal{E}')$;
- $\forall e, e' \in C \circ$ *if* $\mu(e) = \mu(e')$ *then* $e = e'$;
- $\forall e \in C \circ Eot'(\mu(e)) \leq Eot(e)$ *and* $Lot(e) \leq Lot'(\mu(e))$.

Timed event structures and their morphisms form a category of timed event structures, **TES**.

Lemma 3. *Given a morphism $(\mu, \lambda) : \mathcal{E} \to \mathcal{E}'$ of* **TES**, *if $TC = (C,T)$ is a timed configuration of $\mathcal{E}$, then $TC' = (\mu\ C, T')$, where $T' \circ \mu = T$ will be a timed configuration of $\mathcal{E}'$.*

3.5. Timed Event Trees. The main goal of this paper is to expose an adjunction from the category of timed causal trees to the category of timed event structures. In order to achieve this aim, we will use a larger model, timed event trees, that embeds timed causal trees as well as timed event structures.

Definition 17. *A* timed event tree *$\mathcal{ET}$ is a tuple $(S, s_{in}, E, T, <, L, l, Eot, Lot)$, where (S, s_{in}, E, T) is a timed synchronization tree, $<\subseteq E \times E$ is a strict order, L is a set of labels, $l : E \to L$ is a labelling function, and $Eot,\ Lot : E \to \mathbf{R}$ are functions of the earliest and latest occurrence times of events, satisfying the following:*

(i) for all $e \in E$ there exists a transition $(s, e, eot, lot, s') \in T$;

(ii) if $s \xrightarrow[eot,\ lot]{e} s'$ and $s \xrightarrow[eot',\ lot']{e} s''$, then $eot = eot'$, $lot = lot'$ and $s' = s''$;

(iii) if $s \xrightarrow[eot,\ lot]{e} s'$ and $u \xrightarrow[eot',\ lot']{e} u'$, then there is no sequence $s_0 \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \dots s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ $(k \geq 0)$ such that $(s' = s_0 \wedge u = s_k)$ or $(u' = s_0 \wedge s = s_k)$;

(iv) if $e < e'$ and $s \xrightarrow[eot,\ lot]{e'} s'$ then there is a sequence $s_0 \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \dots s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ $(k \geq 0)$ such that $e_1 = e$ and $s = s_k$;

(v) $Eot(e) \leq Lot(e)$, for all $e \in E$;

(vi) $Eot(e) \le eot \le lot \le Lot(e)$, *for all* $(s, e, eot, lot, s_1) \in T$.

We say two events e, e' of a timed event tree $\mathcal{ET}$ are *consistent* (denoted $e\ Con_{\mathcal{ET}}\ e'$) iff $e = e'$ or there exists a sequence $s_0 \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ \dots\ s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ $(k \ge 0)$ such that ($e_1 = e$ and $e_k = e'$) or ($e_1 = e'$ and $e_k = e$).

A *run* of $\mathcal{ET}$ is a sequence $\gamma = \langle s_0, \nu_0\rangle \stackrel{e_1}{\to} \langle s_1, \nu_1\rangle \ \dots\ \langle s_{n-1}, \nu_{n-1}\rangle \stackrel{e_n}{\to} \langle s_n, \nu_n\rangle$ such that $\nu_0 \le \nu_1 \le \dots \le \nu_n$ and for all $1 \le i \le n$ there is a transition $s_{i-1} \xrightarrow[eot_i,\ lot_i]{e_i} s_i$ such that $eot_i \le \nu_i \le lot_i$. Here, $s_0 = s_{in}$ and ν_0 is defined to be 0.

Lemma 4. *Let* $\mathcal{ET} = (S, s_{in}, E, T, <, L, l, Eot, Lot)$ *is a timed event tree and* $s_{in} \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ \dots\ s_{n-1} \xrightarrow[eot_n,\ lot_n]{e_n} s_n$ *for some* $n \ge 1$. *Then* $e_n{\downarrow} \subseteq \{e_1, \dots, e_n\}$.

Now let us define the category of timed event trees.

Definition 18. *Let* $\mathcal{ET} = (S, s_{in}, E, T, <, L, l, Eot, Lot)$ *and* $\mathcal{ET}' = (S', s'_{in}, E', T', <', L', l', Eot', Lot')$ *be timed event trees. A pair* (μ, λ), *where* $\mu : E \to E'$ *and* $\lambda : L \to L'$ *are functions, is called a* morphism, *iff*

(i) $\mu(e){\downarrow} \subseteq \mu(e{\downarrow})$;

(ii) $l' \circ \mu = \lambda \circ l$;

(iii) *if* $s_{in} \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ \dots\ s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ $(k \ge 0)$, *then* $s'_{in} \xrightarrow[eot'_1,\ lot'_1]{\mu(e_1)} s'_1 \ \dots\ s'_{k-1} \xrightarrow[eot'_k,\ lot'_k]{\mu(e_k)} s'_k$ *for some* $s'_j \in S'$ *and* $eot'_j \le eot_j$ *and* $lot_j \le lot'_j$ $(1 \le j \le k)$;

(iv) $Eot'(\mu(e)) \le Eot(e)$ *and* $Lot(e) \le Lot'(\mu(e))$, *for all* $e \in E$.

Lemma 5. *Given timed event trees* $\mathcal{ET} = (S, s_{in}, E, T, <, L, l, Eot, Lot)$ *and* $\mathcal{ET}' = (S', s'_{in}, E', T', <', L', l', Eot', Lot')$, *a morphism* $(\mu, \lambda) : \mathcal{ET} \to \mathcal{ET}'$ *generates the unique function* $\sigma_\mu : S \to S'$ *such that* (σ_μ, μ) *is a morphism between* (S, s_{in}, E, T) *and* (S', s'_{in}, E', T'), *and preserves concurrency: for all* $e, e' \in E$ *if* $e\ Con_{\mathcal{ET}}\ e'$ *and* $\mu(e) <' \mu(e')$, *then* $e < e'$.

Timed event trees and morphisms between them form the *category of timed event trees*, **TET**.

Lemma 6. *Given a morphism* $(\mu, \lambda) : \mathcal{ET} \to \mathcal{ET}'$ *of* **TET**, *if we have a run* $\gamma = \langle s_0, \nu_0\rangle \stackrel{e_1}{\to} \langle s_1, \nu_1\rangle \ \dots\ \langle s_{n-1}, \nu_{n-1}\rangle \stackrel{e_n}{\to} \langle s_n, \nu_n\rangle$ *of* $\mathcal{T}$ *then* $\gamma' = \langle \sigma_\mu(s_0), \nu_0\rangle \stackrel{\mu(e_1)}{\to} \langle \sigma_\mu(s_1), \nu_1\rangle \ \dots\ \langle \sigma_\mu(s_{n-1}), \nu_{n-1}\rangle \stackrel{\mu(e_n)}{\to} \langle \sigma_\mu(s_n), \nu_n\rangle$ *will be the run of* $\mathcal{ET}'$.

4. Relations Between Timed Models for Concurrency. In this section we investigate how the category of timed causal trees relates to the other timed model categories. In particular, we show that there is a coreflection from timed synchronization trees to timed causal trees, a

coreflection from timed synchronization trees to timed event structures, a coreflection from timed causal trees to timed event trees, and a reflection from timed event trees to timed event structures. Thus, we will get the adjunction from timed causal trees to timed event structures which arises as the composition of a coreflection from timed causal trees to timed event trees and a reflection from timed event trees to timed event structures.

4.1. A coreflection between the categories TCT and TST. First, we investigate a relation between the categories **TCT** and **TST**. Clearly, any timed causal tree is a timed synchronization tree. Hence, we have a functor $\mathbf{c2s} : \mathbf{TCT} \to \mathbf{TST}$ that forgets about the causality information and keeps morphisms. Moreover, it is easy to see that **c2s** is a faithful functor.

On the other hand, every timed synchronization tree determines a timed causal tree, in which the causal dependency relation is given by the order of the transitions in the tree. Now we can define a functor $\mathbf{s2c} : \mathbf{TST} \to \mathbf{TCT}$.

Definition 19. *Let* $\mathcal{S} = (S,\ s_{in},\ L,\ T)$ *and* $\mathcal{S}' = (S',\ s'_{in},\ L',\ T')$ *be timed synchronization trees and* $(\mu, \lambda) : \mathcal{S} \to \mathcal{S}'$ *be a morphism of* **TST**. *Define* $\mathbf{s2c}(\mathcal{S}) = (S,\ s_{in},\ L,\ T, <^*)$, *where* $(s, a, eot, lot, s') <^* (u,\ b,\ eot',\ lot',\ u')$ *if and only if there exists a sequence of transitions* $s' \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ \dots\ s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ *for some* $k \geq 1$ *such that* $s_k = u$; *and define* $\mathbf{s2c}(\mu, \lambda) = (\mu, \lambda)$.

Proposition 1. *The mapping* **s2c** *is a fully faithful functor.*

Доказательство. First, we note that $\mathbf{s2c}(\mathcal{S})$ is a timed causal tree for all timed synchronization trees $\mathcal{S} = (S,\ s_{in},\ L,\ T)$.

Second, we should check that $\mathbf{s2c}(\mu, \lambda) = (\mu, \lambda)$ is a morphism of **TCT** for all morphisms $(\mu, \lambda) : \mathcal{S} \to \mathcal{S}'$ of **TST**. We only need to prove that μ preserves concurrency. Let (s, a, eot, lot, s'), $(u, b, eot', lot', u') \in T$, $(s,\ a,\ eot,\ lot,\ s')\ Con\ (u,\ b,\ eot',\ lot',\ u')$ and $(\mu(s),\ \lambda(a),\ eot',\ lot',\ \mu(s')) <'^* (\mu(u),\ \lambda(b),\ eot',\ lot',\ u')$. This implies the existence of a sequence $\mu(s') \xrightarrow[eot'_1,\ lot'_1]{e'_1} s'_1 \ \dots\ s'_{k-1} \xrightarrow[eot'_k,\ lot'_k]{e'_k} s'_k = \mu(u)$ for some $k \geq 1$. Hence, $(s, a, eot, lot, s') \neq (u, b, eot', lot', u')$, by the item (ii) of Definition 12. Furthermore, since $(s,\ a,\ eot,\ lot,\ s')\ Con\ (u,\ b,\ eot',\ lot',\ u')$, we may conclude that either $(s, a, eot, lot, s') <^* (u, b, eot', lot', u')$ or $(u, b, eot', lot', u') <^* (s, a, eot, lot, s')$. Assume $(u,\ b,\ eot',\ lot',\ u') <^* (s,\ a,\ eot,\ lot,\ s')$. This means that there exists a sequence $u' \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ \dots\ s_{l-1} \xrightarrow[eot_l,\ lot_l]{e_l} s_l = s$ for some $l \geq 1$. This implies that $\mu(u') \xrightarrow[eot''_1,\ lot''_1]{\lambda(e_1)} \mu(s_1) \ \dots\ \mu(s_{l-1}) \xrightarrow[eot''_l,\ lot''_l]{\lambda(e_l)} \mu(s_l) = \mu(s)$. This contradicts the item (ii) of Definition 12. Thus, $(s, a, eot, lot, s') <^* (u, b, eot', lot', u')$.

Third, consider an identity morphism $(1_S, 1_L) : \mathcal{S} \to \mathcal{S}$ and a pair of morphisms $(\sigma, \lambda) : \mathcal{S} \to \mathcal{S}'$ and $(\sigma', \lambda') : \mathcal{S}' \to \mathcal{S}''$ from **TST**. It is obvious that $\mathbf{s2c}(1_S,\ 1_L) = (1_S,\ 1_L)$ and $\mathbf{s2c}((\sigma', \lambda') \circ (\sigma, \lambda)) = \mathbf{s2c}(\sigma' \circ \sigma, \lambda' \circ \lambda) = (\sigma' \circ \sigma, \lambda' \circ \lambda) = \mathbf{s2c}(\sigma', \lambda') \circ \mathbf{s2c}(\sigma, \lambda)$. Thus, **s2c** is indeed a functor.

Finally, we need to clarify that **s2c** is a fully faithful functor. Take arbitrary objects $\mathcal{S}$ and $\mathcal{S}'$ of **TST**. Define a function $F_{\mathcal{S},\mathcal{S}'} : \mathbf{TST}(\mathcal{S},\mathcal{S}') \to \mathbf{TCT}(\mathbf{s2c}(\mathcal{S}),\mathbf{s2c}(\mathcal{S}'))$ such that $F_{\mathcal{S},\mathcal{S}'}(\sigma,\lambda) = \mathbf{s2c}(\sigma,\lambda) = (\sigma,\lambda)$ for all morphisms $(\sigma,\lambda) : \mathcal{S} \to \mathcal{S}'$ of **TST**. Since **s2c** is a functor, $F_{\mathcal{S},\mathcal{S}'}$ is a function. Moreover, it is easy to check that $F_{\mathcal{S},\mathcal{S}'}$ is injective, because $F_{\mathcal{S},\mathcal{S}'}(\sigma,\lambda) = (\sigma,\lambda)$. Hence, **s2c** is a faithful functor. Next, take an arbitrary morphism $(\sigma,\lambda) : \mathbf{s2c}(\mathcal{S}) \to \mathbf{s2c}(\mathcal{S}')$ of **TCT**. Clearly, (σ,λ) is a morphism of **TST** from $\mathcal{S}$ to $\mathcal{S}'$ and $F_{\mathcal{S},\mathcal{S}'}(\sigma,\lambda) = (\sigma,\lambda)$. Thus, **s2c** is a full functor. □

Proposition 2. *Let $\mathcal{S} = (S,\ s_{in},\ L,\ T)$ be a timed synchronization tree. Then $\mathbf{s2c}(\mathcal{S})$ is a timed causal tree, $(1_S,1_L) : \mathcal{S} \to \mathbf{c2s}(\mathbf{s2c}(\mathcal{S}))$ is an isomorphism and the pair $(\mathbf{s2c}(\mathcal{S}),(1_S,1_L))$ is a reflection of $\mathcal{S}$ along* **c2s**.

Доказательство. It is clear that $\mathbf{c2s}(\mathbf{s2c}(\mathcal{S})) = \mathcal{S}$. Hence, $(1_S,1_L) : \mathcal{S} \to \mathbf{c2s}(\mathbf{s2c}(\mathcal{S})) = \mathcal{S}$ is a morphism of **TST**. Moreover, it is an isomorphism.

Now we should prove that $(\mathbf{s2c}(\mathcal{S}),(1_S,1_L))$ is a reflection of $\mathcal{S}$ along **c2s**, i.e. whenever $\mathcal{C}'$ is a timed causal tree and $(\sigma,\lambda) : \mathcal{S} \to \mathbf{c2s}(\mathcal{C}')$ is a morphism of **TST**, then there exists a unique morphism $(g,\lambda') : \mathbf{s2c}(\mathcal{S}) \to \mathcal{C}'$ such that $(\sigma,\lambda) = \mathbf{c2s}(g,\lambda') \circ (1_S,1_L)$. Since $\mathbf{c2s}(g,\lambda') = (g,\lambda')$, we may conclude that λ' must be equal to λ and g must match σ. Hence, we should only show that $(\sigma,\lambda) : \mathbf{s2c}(\mathcal{S}) \to \mathcal{C}'$ is a morphism of **TCT**. Since $(\sigma,\lambda) : \mathcal{S} \to \mathbf{c2s}(\mathcal{C}')$ is a morphism of **TST**, we only need to check that σ preserves concurrency. Take an arbitrary (s,a,eot,lot,s'), $(u,b,eot^*,lot^*,u') \in T$ such that $(s,a,eot,lot,s')\ Con\ (u,b,eot^*,lot^*,u')$ and $(\sigma(s),\lambda(a),eot',lot',\sigma(s')) <' (\sigma(u),\lambda(b),eot'^*,lot'^*,\sigma(u'))$. Since $\mathcal{C}'$ is a timed causal tree, we may conclude that there exists a sequence $\sigma(s') \xrightarrow[eot'_1,\ lot'_1]{e'_1} \bar{s}_1 \ldots \bar{s}_{k-1} \xrightarrow[eot'_k,\ lot'_k]{e'_k} \sigma(u)$ for some $k \geq 1$.

Since $(s,a,eot,lot,s')\ Con\ (u,b,eot^*,lot^*,u')$, we have three admissible cases: $(s,\ a,\ eot,\ lot,\ s') = (u,\ b,\ eot^*,\ lot^*,\ u')$, $(s,\ a,\ eot,\ lot,\ s') <^* (u,\ b,\ eot^*,\ lot^*,\ u')$ and $(u,\ b,\ eot^*,\ lot^*,\ u') <^* (s,\ a,\ eot,\ lot,\ s')$. If $(s,\ a,\ eot,\ lot,\ s') = (u,\ b,\ eot^*,\ lot^*,\ u')$ then $(\sigma(s),\ \lambda(a),\ eot',\ lot',\ \sigma(s')) = (\sigma(u),\ \lambda(b),\ eot'^*,\ lot'^*,\ \sigma(u'))$, that contradicts our conditions. If $(u,\ b,\ eot^*,\ lot^*,\ u') <^* (s,\ a,\ eot,\ lot,\ s')$, we have a sequence $u' \xrightarrow[eot_1,\ lot_1]{e_1} \tilde{s}_1 \ldots \tilde{s}_{m-1} \xrightarrow[eot_m,\ lot_m]{e_m} s$ for some $m \geq 1$. Hence, $\sigma(u') \xrightarrow[eot''_1,\ lot''_1]{\lambda(e_1)} \sigma(\tilde{s}_1) \ldots \sigma(\tilde{s}_{m-1}) \xrightarrow[eot''_m,\ lot''_m]{\lambda(e_m)} \sigma(s)$. This contradicts the item (ii) of Definition 12. Hence, $(s,a,eot,lot,s') <^* (u,b,eot^*,lot^*,u')$.

Thus we can conclude that $(\mathbf{s2c}(\mathcal{S}),(1_S,1_L))$ is a reflection of $\mathcal{S}$ along **c2s**. □

The above results enable us to exhibit an adjunction between the categories **TST** and **TCT**.

Theorem 1. *The functor* **c2s** *is right adjoint to* **s2c** *and this adjunction is a coreflection.*

Доказательство. The first assertion follows from Proposition 2 and from the fact that for all morphisms $(\sigma,\lambda) : \mathcal{C} = (S,\ s_{in},\ L,\ T,<) \to \mathcal{C}' = (S',\ s'_{in},\ L',\ T',<')$ it is true that $(1_{S'},1_{L'}) \circ$

$(\sigma,\lambda) = (\sigma,\lambda) = \mathbf{s2c}(\mathbf{c2s}(\sigma,\lambda)) = \mathbf{s2c}(\mathbf{c2s}(\sigma,\lambda))\circ(1_S,1_L)$. Moreover, it follows from Proposition 2 that the unit ψ associates each timed synchronization tree $\mathcal{S} = (S,\ s_{in},\ L,\ T)$ with the isomorphism $(1_S,1_L):\mathcal{S}\to\mathbf{c2s}(\mathbf{s2c}(\mathcal{S}))$. Hence, ψ is a natural isomorphism. □

Thus, **TST** embeds fully and faithfully into **TCT** and is equivalent to the full subcategory of **TCT** consisting of those timed causal trees $\mathcal{C}$ that are isomorphic to $\mathbf{s2c}(\mathbf{c2s}(\mathcal{C}))$.

4.2. A coreflection between the categories TCT and TET. In this subsection we establish that there is a coreflection from timed causal trees to timed event trees. Note that any timed event tree gives rise to a timed causal tree by forgetting about events. Hence, we can specify a functor $\mathbf{et2c}:\mathbf{TET}\to\mathbf{TCT}$.

Definition 20. *Let* $\mathcal{ET} = (S,s_{in},E,T,<,L,l,Eot,Lot)$ *and* $\mathcal{ET}' = (S',\ s'_{in},\ E',\ T',\ <',\ L',\ l',\ Eot',\ Lot')$ *be timed event trees and* $(\mu,\lambda):\mathcal{ET}\to\mathcal{ET}'$ *be a morphism of* **TET**. *Define* $\mathbf{et2c}(\mathcal{ET}) = (S,s_{in},L,T^*,<^*)$, *where* $T^* = \{(s,l(e),eot,lot,s') \mid (s,e,eot,lot,s')\in T\}$, $(s,l(e),eot,lot,s') <^* (u,\ l(e'),\ eot',\ lot',\ u')$ *if and only if* $e<e'$ *and there exists a sequence* $s' \xrightarrow[eot_1,\ lot_1]{e_1} s_1\ \ldots\ s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} u$ *for some* $k\geq 0$; *and define* $\mathbf{et2c}(\mu,\lambda) = (\sigma_\mu,\lambda)$, *where* $\sigma_\mu:S\to S'$ *is defined by* μ *as in Lemma 5.*

Proposition 3. *The mapping* **et2c** *is a faithful functor.*

Доказательство. It is clear that $\mathbf{et2c}(\mathcal{ET})$ is indeed a timed causal tree for all timed event trees $\mathcal{ET}$. The fact that $\mathbf{et2c}(\mu,\lambda) = (\sigma_\mu,\lambda)$ is a morphism of **TCT** for all morphisms $(\mu,\lambda):\mathcal{ET}\to\mathcal{ET}'$ of **TET** follows from Lemma 5 and the equation $\lambda\circ l = l'\circ\mu$. Next, we consider an identity morphism $(1_E,1_L):\mathcal{ET}\to\mathcal{ET}$ and a pair of morphisms $(\mu,\lambda):\mathcal{ET}\to\mathcal{ET}'$ and $(\mu',\lambda'):\mathcal{ET}'\to\mathcal{ET}''$ from **TET**. Obviously, $\mathbf{et2c}(1_E,1_L) = (\sigma_{1_E},1_L) = (1_S,1_L)$, where $(1_S,1_L):\mathbf{et2c}\mathcal{ET}\to\mathbf{et2c}\mathcal{ET}$ is an identity morphism of **TCT**, and $\mathbf{et2c}((\mu',\lambda')\circ(\mu,\lambda)) = \mathbf{et2c}(\mu'\circ\mu,\lambda'\circ\lambda) = (\sigma_{\mu'\circ\mu},\lambda'\circ\lambda) = (\sigma_{\mu'}\circ\sigma_\mu,\lambda'\circ\lambda) = \mathbf{et2c}(\mu',\lambda')\circ\mathbf{et2c}(\mu,\lambda)$. Hence, we can conclude that **et2c** is a functor.

Now we need to show that the functor **et2c** is faithful. Take an arbitrary pair of objects $\mathcal{ET}$ and $\mathcal{ET}'$ of **TET**. Define a function $F_{\mathcal{ET},\mathcal{ET}'}:\mathbf{TET}(\mathcal{ET},\mathcal{ET}')\to\mathbf{TCT}(\mathbf{et2c}(\mathcal{ET}),\mathbf{et2c}(\mathcal{ET}'))$ such that $F_{\mathcal{ET},\mathcal{ET}'}(\mu,\lambda) = \mathbf{et2c}(\mu,\lambda) = (\sigma_\mu,\lambda)$ for all morphisms $(\mu,\lambda):\mathcal{ET}\to\mathcal{ET}'$ of **TET**. Clearly, $F_{\mathcal{ET},\mathcal{ET}'}$ is indeed a function, because **et2c** is a functor. Check that $F_{\mathcal{ET},\mathcal{ET}'}$ is injective. Take arbitrary two morphisms $(\mu_1,\lambda_1):\mathcal{ET}\to\mathcal{ET}'$ and $(\mu_2,\lambda_2):\mathcal{ET}\to\mathcal{ET}'$ such that $F_{\mathcal{ET},\mathcal{ET}'}(\mu_1,\lambda_1) = F_{\mathcal{ET},\mathcal{ET}'}(\mu_2,\lambda_2)$. This implies that $(\sigma_{\mu_1},\lambda_1) = (\sigma_{\mu_2},\lambda_2)$. Hence, $\lambda_1 = \lambda_2$ and $\sigma_{\mu_1} = \sigma_{\mu_2}$. Since σ_{μ_1} defines the function μ_1 in a unique way, we may conclude that $\mu_1 = \mu_2$. Hence, $F_{\mathcal{ET},\mathcal{ET}'}$ is injective, i.e. **et2c** is a faithful functor. □

Note, every timed causal tree $\mathcal{C}$ determines a timed event tree which is induced by $\mathcal{C}$ when we assume that each transition of $\mathcal{C}$ represents a separate event. This means that we take

the transitions of $\mathcal{C}$ as events , and label each arc of $\mathcal{C}$ by the corresponding transition. This operation can be easily extended to a functor $\mathbf{c2et} : \mathbf{TCT} \to \mathbf{TET}$.

Definition 21. *Let* $\mathcal{C} = (S,\ s_{in},\ L,\ T, <)$ *and* $\mathcal{C}' = (S',\ s'_{in},\ L',\ T', <')$ *be timed causal trees and* $(\sigma, \lambda) : \mathcal{C} \to \mathcal{C}'$ *be a morphism of* **TCT**. *Define* $\mathbf{c2et}(\mathcal{C}) = (S, s_{in}, T, T^\star, <, L, l, Eot, Lot)$, *where* $T^\star = \{(s, (s, a, eot, lot, s'), eot, lot, s') \mid (s,\ a,\ eot,\ lot,\ s') \in T\}$, $l(s, a, eot, lot, s') = a$, $Eot(s, a, eot, lot, s') = eot$ *and* $Lot(s, a, eot, lot, s') = lot$; *and define* $\mathbf{c2et}(\sigma, \lambda) = (\mu, \lambda)$, *where* $\mu : T \to T'$ *is given by the following equality:* $\mu(s, a, eot, lot, s') = (\sigma(s), \lambda(a), eot', lot', \sigma(s')) \in T'$ *for some* $eot' \le eot$ *and* $lot \le lot'$.

Proposition 4. *The mapping* **c2et** *is a faithful functor.*

Доказательство. It is easy to check that $\mathbf{c2et}(\mathcal{C}) = (S, s_{in}, T, T^\star, <, L, l, Eot, Lot)$ is a timed event tree for all timed causal trees $\mathcal{C} = (S,\ s_{in},\ L,\ T, <)$.

Now, we need to prove that $\mathbf{c2et}(\sigma, \lambda) : \mathbf{c2et}(\mathcal{C}) \to \mathbf{c2et}(\mathcal{C}')$ is a morphism of **TET** for all morphisms $(\sigma, \lambda) : \mathcal{C} \to \mathcal{C}'$ of **TCT**. W.l.o.g. assume that $\mathcal{C} = (S,\ s_{in},\ L,\ T, <)$ and $\mathcal{C}' = (S',\ s'_{in},\ L',\ T', <')$. Then, $\mathbf{c2et}(\mathcal{C}) = (S,\ s_{in},\ T,\ T^\star,\ <,\ L,\ l,\ Eot,\ Lot)$ and $\mathbf{c2et}(\mathcal{C}') = (S',\ s'_{in},\ T',\ T'^\star,\ <',\ L',\ l',\ Eot',\ Lot')$, where $T^\star = \{(s,\ (s, a, eot, lot, s'),\ eot,\ lot,\ s') \mid (s, a, eot, lot, s') \in T\}$, $l(s, a, eot, lot, s') = a$, $Eot(s, a, eot, lot, s') = eot$, $Lot(s, a, eot, lot, s') = lot$, $T'^\star = \{(u',\ (u', a', eot', lot', u''),\ eot',\ lot',\ u'') \mid (u', a', eot', lot', u'') \in T'\}$, $l'(u', a', eot', lot', u'') = a'$, $Eot'(u', a', eot', lot', u'') = eot'$ and $Lot'(u',\ a',\ eot',\ lot',\ u'') = lot'$. Moreover, $\mathbf{c2et}(\sigma, \lambda) = (\mu, \lambda)$, where μ associates $(s,\ a,\ eot,\ lot,\ s') \in T$ with some transition $(\sigma(s), \lambda(a), eot', lot', \sigma(s'))$ of $\mathbf{c2et}(\mathcal{C}')$ with $eot' \le eot$ and $lot \le lot'$. The existence and unicity of such transition follows from the item (ii) of Definition 11 and the item (iii) of Definition 12. Hence, $\mu : T \to T'$ and $\lambda : L \to L'$ are functions. Check that (μ, λ) satisfies the requirements of Definition 18.

(i) Let us show that $\mu(s, a, eot, lot, s')\downarrow \subseteq \mu((s, a, eot, lot, s')\downarrow)$.

Take an arbitrary $(s, a, eot, lot, s') \in T$. Using the items (i), (iii) of Definition 12, we can find a unique sequence $s_{in} \xrightarrow[eot_1,\ lot_1]{a_1} s_1 \ldots s_{k-1} \xrightarrow[eot_k,\ lot_k]{a_k} s_k = s$ for some $k \ge 0$. Since (σ, λ) is a morphism of **TCT**, we have $\sigma(s_{in}) = s'_{in} \xrightarrow[eot'_1,\ lot'_1]{\lambda(a_1)} \sigma(s_1) \ldots \sigma(s_{k-1}) \xrightarrow[eot'_k,\ lot'_k]{\lambda(a_k)} \sigma(s_k) = \sigma(s) \xrightarrow[eot',\ lot']{\lambda(a)} \sigma(s')$ for some $eot'_1, \ldots, eot'_k, eot', lot'_1, \ldots, lot'_k, lot' \in \mathbf{R}$ such that $eot' \le eot$, $lot \le lot'$ and $eot'_j \le eot_j$ and $lot_j \le lot'_j$ for all $1 \le j \le k$. Clearly, $\mu(s, a, eot, lot, s') = (\sigma(s), \lambda(a), eot', lot', \sigma(s'))$. Since $\mathbf{c2et}(\mathcal{C}')$ is a timed event tree, we have $\mu(s, a, eot, lot, s')\downarrow \subseteq \{(\sigma(s_{in}) = s'_{in}, \lambda(a_1), eot'_1, lot'_1, \sigma(s_1)), \ldots, (\sigma(s_{k-1}), \lambda(a_k), eot'_k, lot'_k, \sigma(s))\}$ by Lemma 4. Hence, if $e' <' \mu(s,\ a,\ eot,\ lot,\ s')$ then $e' = (\sigma(s_{j-1}),\ \lambda(a_j),\ eot'_j,\ lot'_j,\ \sigma(s_j)) = \mu(s_{j-1},\ a_j,\ eot_j,\ lot_j,\ s_j)$ for some $1 \le j \le k$. This implies $\mu(s_{j-1},\ a_j,\ eot_j,\ lot_j,\ s_j) <' \mu(s,\ a,\ eot,\ lot,\ s')$. According to Definition 14, it is easy to see that $(s_{j-1},\ a_j,\ eot_j,\ lot_j,\ s_j)$

$< (s,\ a,\ eot,\ lot,\ s')$. Thus, $(s_{j-1},\ a_j,\ eot_j,\ lot_j,\ s_j) \in (s,a,\ eot,\ lot,\ s')\downarrow$. Furthermore, $\mu(s,a,eot,lot,s')\downarrow \subseteq \mu((s,a,eot,lot,s')\downarrow)$.

(ii) It is clear that $l' \circ \mu(s,a,eot,lot,s') = l'(\sigma(s),\ \lambda(a),\ eot',\ lot',\ \sigma(s')) = \lambda(a) = \lambda \circ l(s,\ a,\ eot,\ lot,\ s')$ for all $(s,a,eot,lot,s') \in T$.

(iii) Let $s_{in} \xrightarrow[eot_1,\ lot_1]{(s_{in},a_1,eot_1,lot_1,s_1)} s_1 \ldots s_{k-1} \xrightarrow[eot_k,\ lot_k]{(s_{k-1},a_k,eot_k,lot_k,s_k)} s_k$ $(k \geq 0)$ in $\mathbf{c2et}(\mathcal{C})$. This means that $s_{in} \xrightarrow[eot_1,\ lot_1]{a_1} s_1 \ldots s_{k-1} \xrightarrow[eot_k,\ lot_k]{a_k} s_k$ in $\mathcal{C}$. Since (σ,λ) is a morphism of $\mathbf{TCT}$, we may conclude that $\sigma(s_{in}) = s'_{in} \xrightarrow[eot'_1,\ lot'_1]{\lambda(a_1)} \sigma(s_1) \ldots \sigma(s_{k-1}) \xrightarrow[eot'_k,\ lot'_k]{\lambda(a_k)} \sigma(s_k)$ in $\mathcal{C}'$ for some $eot'_1, \ldots, eot'_k$, $lot'_1, \ldots, lot'_k \in \mathbf{R}$ such that $eot'_j \leq eot_j$ and $lot_j \leq lot'_j$ for all $1 \leq j \leq k$. Hence, $\sigma(s_{in}) = s'_{in} \xrightarrow[eot'_1,\ lot'_1]{(\sigma(s_{in}),\lambda(a_1),eot'_1,lot'_1,\sigma(s_1))} \sigma(s_1) \ldots \sigma(s_{k-1}) \xrightarrow[eot'_k,\ lot'_k]{(\sigma(s_{k-1}),\lambda(a_k),eot'_k,lot'_k,\sigma(s_k))} \sigma(s_k)$ in $\mathbf{c2et}(\mathcal{C}')$ and for all $1 \leq j \leq k$ it holds that $\mu(s_{j-1},a_j,eot_j,lot_j,s_j) = (\sigma(s_{j-1}),\ \lambda(a_j),\ eot'_j,\ lot'_j,\ \sigma(s_j))$.

(iv) Obviously, $Eot'(\mu(s,a,eot,lot,s')) = eot' \leq eot = Eot(s,a,eot,lot,s')$ and $Lot(s,\ a,\ eot,\ lot,\ s') = lot \leq lot' = Lot'(\mu(s,a,eot,lot,s'))$ for all $(s,\ a,\ eot,\ lot,\ s') \in T$.

This means that (μ,λ) is indeed a morphism of $\mathbf{TET}$ from $\mathbf{c2et}(\mathcal{C})$ to $\mathbf{c2et}(\mathcal{C}')$.

Next, we consider an identity morphism $(1_S,1_L) : \mathcal{C} \to \mathcal{C}$ and a pair of morphisms $(\sigma,\lambda) : \mathcal{C} \to \mathcal{C}'$ and $(\sigma',\lambda') : \mathcal{C}' \to \mathcal{C}''$ from $\mathbf{TCT}$. Clearly, $\mathbf{c2et}(1_S,1_L) = (\mu_{1_S,1_L},1_L) = (1_T,1_L)$, where $(1_T,1_L) : \mathbf{c2et}\mathcal{C} \to \mathbf{c2et}\mathcal{C}$ is an identity morphism of $\mathbf{TET}$, and $\mathbf{c2et}((\sigma',\lambda')\circ(\sigma,\lambda)) = \mathbf{c2et}(\sigma' \circ \sigma, \lambda' \circ \lambda) = (\mu_{\sigma' \circ \sigma,\lambda' \circ \lambda}, \lambda' \circ \lambda) = (\mu_{\sigma',\lambda'} \circ \mu_{\sigma,\lambda}, \lambda' \circ \lambda) = \mathbf{c2et}(\sigma',\lambda') \circ \mathbf{c2et}(\sigma,\lambda)$. Thus, $\mathbf{c2et}$ is indeed a functor.

In conclusion we prove that the functor $\mathbf{c2et}$ is faithful. Take an arbitrary pair of timed causal trees $\mathcal{C}$ and $\mathcal{C}'$. Define a function $F_{\mathcal{C},\mathcal{C}'} : \mathbf{TCT}(\mathcal{C},\mathcal{C}') \to \mathbf{TET}(\mathbf{c2et}(\mathcal{C}),\mathbf{c2et}(\mathcal{C}'))$ such that $F_{\mathcal{C},\mathcal{C}'}(\sigma,\lambda) = \mathbf{c2et}(\sigma,\lambda) = (\mu_{\sigma,\lambda},\lambda)$ for all morphisms $(\sigma,\lambda) : \mathcal{C} \to \mathcal{C}'$ of $\mathbf{TCT}$. It is easy to see that $F_{\mathcal{C},\mathcal{C}'}$ is indeed a function, because $\mathbf{c2et}$ is a functor. Verify that $F_{\mathcal{C},\mathcal{C}'}$ is an injective function. Take arbitrary two morphisms $(\sigma_1,\lambda_1) : \mathcal{C} \to \mathcal{C}'$ and $(\sigma_2,\lambda_2) : \mathcal{C} \to \mathcal{C}'$ such that $F_{\mathcal{C},\mathcal{C}'}(\sigma_1,\lambda_1) = F_{\mathcal{C},\mathcal{C}'}(\sigma_2,\lambda_2)$. This implies $(\mu_{\sigma_1,\lambda_1},\lambda_1) = (\mu_{\sigma_2,\lambda_2},\lambda_2)$. Hence, $\lambda_1 = \lambda_2$ and $\mu_{\sigma_1,\lambda_1} = \mu_{\sigma_2,\lambda_2}$. Contemplate an arbitrary state $s \in S$. Since $\mathcal{C}$ is a timed synchronization tree, we have some transition (s',a,eot,lot,s) of $\mathcal{C}$. Clearly, for all $i = 1,2$ $\mu_{\sigma_i,\lambda_i}(s',a,eot,lot,s) = (\sigma_i(s'),\ \lambda_i(a),\ eot_i,\ lot_i,\ \sigma_i(s)) \in T'$. Since $\mu_{\sigma_1,\lambda_1} = \mu_{\sigma_2,\lambda_2}$, we have $(\sigma_1(s'),\ \lambda_1(a),\ eot_1,\ lot_1,\ \sigma_1(s)) = (\sigma_2(s'),\ \lambda_2(a),\ eot_2,\ lot_2,\ \sigma_2(s))$. Hence, $\sigma_1(s) = \sigma_2(s)$. This fact implies $\sigma_1 = \sigma_2$. Thus, $F_{\mathcal{C},\mathcal{C}'}$ is injective, i.e. $\mathbf{c2et}$ is a faithful functor. □

Proposition 5. *Let $\mathcal{C} = (S,\ s_{in},\ L,\ T,<)$ be a timed causal tree. Then $\mathbf{c2et}(\mathcal{C})$ is a timed event tree, $(1_S,1_L) : \mathcal{C} \to \mathbf{et2c}(\mathbf{c2et}(\mathcal{C}))$ is an isomorphism and the pair $(\mathbf{c2et}(\mathcal{C}),(1_S,1_L))$ is a reflection of $\mathcal{C}$ along $\mathbf{et2c}$.*

Доказательство. Since **c2et** is a functor, $\mathbf{c2et}(\mathcal{C})$ is a timed event tree. Obviously, $\mathbf{c2et}(\mathcal{C}) = (S,\ s_{in},\ T,\ T^{\star},\ <,\ L,\ l,\ Eot,\ Lot)$, where $T^{\star} = \{(s,\ (s,\ a,\ eot,\ lot,\ s'),\ eot,\ lot,\ s') \mid (s,\ a,\ eot,\ lot,\ s') \in T\}$, $l(s,a,eot,lot,s') = a$, $Eot(s,a,eot,lot,s') = eot$ and $Lot(s,a,eot,lot,s') = lot$. Contemplate a timed causal tree $\mathbf{et2c}(\mathbf{c2et}(\mathcal{C}))$. Clearly, $\mathbf{et2c}(\mathbf{c2et}(\mathcal{C})) = (S, s_{in}, L, T^{**}, <^{**})$, where $T^{**} = \{(s, l(e), eot, lot, s') \mid e \in T \text{ and } (s, e, eot, lot, s') \in T^{\star}\}$ and $(s, l(e), eot, lot, s') <^{**} (u, l(e'), eot', lot', u') \iff e, e' \in T,\ e < e'$ and $\exists\ s' \xrightarrow[eot_1,\ lot_1]{(s', a_1, eot_1, lot_1, s_1)} s_1 \ \ldots\ s_{k-1} \xrightarrow[eot_k,\ lot_k]{(s_{k-1}, a_k, eot_k, lot_k, s_k)} s_k = u$ $(k \geq 0)$ in $\mathbf{c2et}(\mathcal{C})$. Hence, $T^{**} = \{(s, l(s,a,eot,lot,s'), eot, lot, s') \mid (s,a,eot,lot,s') \in T\} = T$. Moreover, it holds that $(s,\ l(e),\ eot,\ lot,\ s') <^{**} (u,\ l(e') = b,\ eot',\ lot',\ u') \iff e = (s,\ a,\ eot,\ lot,\ s'),\ e' = (u,\ b,\ eot',\ lot',\ u'),\ (s,\ a,\ eot,\ lot,\ s') < (u,\ b,\ eot',\ lot',\ u')$ and $\exists\ s' \xrightarrow[eot_1,\ lot_1]{a_1} s_1 \ \ldots\ s_{k-1} \xrightarrow[eot_k,\ lot_k]{a_k} s_k = u$ $(k \geq 0)$ in $\mathcal{C}$. Because $\mathcal{C}$ is a timed causal tree, we have $(s,\ l((s,a,eot,lot,s')) = a,\ eot,\ lot,\ s') <^{**} (u,\ l((u,b,eot',lot',u')) = b,\ eot',\ lot',\ u') \iff (s,\ a,\ eot,\ lot,\ s') < (u,\ b,\ eot',\ lot',\ u')$, i.e. $<=<^{**}$. Thus, $\mathbf{et2c}(\mathbf{c2et}(\mathcal{C})) = \mathcal{C}$.

Clearly, $(1_S, 1_L) : \mathcal{C} \to \mathbf{et2c}(\mathbf{c2et}(\mathcal{C})) = \mathcal{C}$ is a morphism of **TCT**. Furthermore, it is an isomorphism.

Now we should prove that $(\mathbf{c2et}(\mathcal{C}), (1_S, 1_L))$ is a reflection of $\mathcal{C}$ along **et2c**, i.e. whenever $\mathcal{ET}'$ is a timed event tree and $(\sigma, \lambda) : \mathcal{C} \to \mathbf{et2c}(\mathcal{ET}')$ is a morphism of **TCT**, there exists a unique morphism $(g, \lambda') : \mathbf{c2et}(\mathcal{C}) \to \mathcal{ET}'$ such that $(\sigma, \lambda) = \mathbf{et2c}(g, \lambda') \circ (1_S, 1_L)$. Since $\mathbf{et2c}(g, \lambda') = (\sigma_g, \lambda')$, we may conclude that λ' must be equal to λ and g must be defined so that $\sigma_g = \sigma$.

W.l.o.g. assume that $\mathcal{ET}' = (S', s'_{in}, E', T', <', L', l', Eot', Lot')$ and $(\sigma, \lambda) : \mathcal{C} \to \mathbf{et2c}(\mathcal{ET}')$ is a morphism of **TCT**. Obviously, $\mathbf{et2c}(\mathcal{ET}') = (S',\ s'_{in},\ L',\ T'^{*}, <'^{*})$, where $T'^{*} = \{(u,\ l'(e),\ eot,\ lot,\ u') \mid (u,\ e,\ eot,\ lot,\ u') \in T'\}$ and $(u, l'(e), eot, lot, u') <'^{*} (t, l'(e'), eot', lot', t') \iff e <' e'$ and there exists a sequence $u' \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ldots s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} t$ for some $k \geq 0$. Define a mapping $g : T \to E'$ as follows: $g(s,a,eot,lot,s') = e'$ such that $l'(e') = \lambda(a)$ and $(\sigma(s), e', eot', lot', \sigma(s')) \in T'$ for some $eot', lot' \in \mathbf{R}$ with $eot' \leq eot$ and $lot \leq lot'$.

First, check that g is a function. Let $(s,a,eot,lot,s') \in T$. Since (σ, λ) is a morphism of **TCT**, we have that $(\sigma(s),\ \lambda(a),\ eot',\ lot',\ \sigma(s')) \in T'^{*}$ for some $eot', lot' \in \mathbf{R}$ such that $eot' \leq eot$ and $lot \leq lot'$. This implies that $(\sigma(s),\ e',\ eot',\ lot',\ \sigma(s')) \in T'$ for some $e' \in E'$ such that $l'(e') = \lambda(a)$. Hence, for all $(s,a,eot,lot,s') \in T$ there is an event e' such that $l'(e') = \lambda(a)$ and $(\sigma(s), e', eot', lot', \sigma(s')) \in T'$ for some $eot', lot' \in \mathbf{R}$ with $eot' \leq eot$ and $lot \leq lot'$. Suppose that we have $e', e'' \in E'$ such that $(\sigma(s), e', eot', lot', \sigma(s'))$, $(\sigma(s), e'', eot'', lot'', \sigma(s')) \in T'$, $eot' \leq eot$, $lot \leq lot'$, $eot'' \leq eot$, $lot \leq lot''$ and $l'(e') = l'(e'') = \lambda(a)$. Due to the item (iii) of Definition 12, it holds that $e' = e''$. Thus, g is well defined.

Second, establish that $(g, \lambda) : \mathbf{c2et}(\mathcal{C}) \to \mathcal{ET}'$ is a morphism of **TET**.

- Check that $g(s,a,eot,lot,s')\downarrow \subseteq g((s,a,eot,lot,s')\downarrow)$.

Assume $e'' \in g(s,\, a,\, eot,\, lot,\, s')\downarrow$. It means that $e'' <' e' = g(s,\, a,\, eot,\, lot,\, s')$ with $(\sigma(s),\, e',\, eot',\, lot',\, \sigma(s')) \in T'$ and $l'(e') = \lambda(a)$. Due to the items (i), (iii) of Definition 12, we have a unique sequence $s_{in} \xrightarrow[eot_1,\, lot_1]{a_1} s_1 \ \ldots\ s_{k-1} \xrightarrow[eot_k,\, lot_k]{a_k} s_k = s$ in $\mathcal{C}$. It means that $s_{in} \xrightarrow[eot_1,\, lot_1]{(s_{in},a_1,eot_1,lot_1,s_1)} s_1 \ \ldots\ s_{k-1} \xrightarrow[eot_k,\, lot_k]{(s_{k-1},a_k,eot_k,lot_k,s_k)} s_k = s \xrightarrow[eot,\, lot]{(s,a,eot,lot,s')} s'$ in $\mathbf{c2et}(\mathcal{C})$. From Lemma 4 we get $(s,a,eot,lot,s')\downarrow\subseteq \{(s_{in},a_1,eot_1,lot_1,s_1),\ldots,(s_{k-1},a_k,eot_k,lot_k,s_k = s),(s,a,eot,lot,s')\}$. Since (σ,λ) is a morphism, it holds that $\sigma(s_{in}) = s'_{in} \xrightarrow[eot'_1,\, lot'_1]{\lambda(a_1)} \sigma(s_1)$ $\ldots\ \sigma(s_{k-1}) \xrightarrow[eot'_k,\, lot'_k]{\lambda(a_k)} \sigma(s_k) = \sigma(s) \xrightarrow[eot',\, lot']{\lambda(a)} \sigma(s')$ in $\mathbf{et2c}(\mathcal{ET}')$ for some eot', eot'_1, …, eot'_k, lot', lot'_1, …, $lot'_k \in \mathbf{R}$ such that $eot' \le eot$, $lot \le lot'$, and $eot'_j \le eot_j$ and $lot_j \le lot'_j$ for all $1 \le j \le k$. Hence, $\sigma(s_{in}) = s'_{in} \xrightarrow[eot'_1,\, lot'_1]{e'_1} \sigma(s_1)\ \ldots\ \sigma(s_{k-1}) \xrightarrow[eot'_k,\, lot'_k]{e'_k} \sigma(s_k) = \sigma(s) \xrightarrow[eot',\, lot']{e'} \sigma(s')$ in $\mathcal{ET}'$ for some e', $e'_1,\ldots,e'_k \in E'$ such that $l'(e') = \lambda(a)$, $l'(e'_j) = \lambda(a_j)$ for all $1 \le j \le k$. Moreover, it is easy to see that $g(s,a,eot,lot,s') = e'$, $g(s_{j-1},\, a_j,\, eot_j,\, lot_j,\, s_j) = e'_j$ $(1 \le j \le k)$. Since $\mathcal{ET}'$ is a timed event tree, we may conclude that $g(s,a,eot,lot,s')\downarrow\subseteq \{g(s_{in},a_1,eot_1,lot_1,s_1),\ \ldots,\ g(s_{k-1},\, a_k,\, eot_k,\, lot_k,\, s_k = s)\}$ by Lemma 4. Assume $e'' \in g(s,a,eot,lot,s')\downarrow$. It means that $e'' = g(s_{j-1},a_j,eot_j,lot_j,s_j) = e'_j$ for some $1 \le j \le k$. This implies that $(\sigma(s_{j-1}),e'_j,eot'_j,lot'_j,\sigma(s_j)) <'^{*} (\sigma(s),e',eot',lot',\sigma(s'))$. According to Definition 14, it holds that $(s_{j-1},e_j,eot_j,lot_j,s_j) < (s,e,eot,lot,s')$. Thus, $e'' \in g((s,a,eot,lot,s')\downarrow)$.

- Obviously, $l' \circ g(s,a,eot,lot,s') = l'(e') = \lambda(a) = \lambda \circ l(s,a,eot,lot,s')$.
- Let $s_{in} \xrightarrow[eot_1,\, lot_1]{(s_{in},a_1,eot_1,lot_1,s_1)} s_1 \ \ldots\ s_{k-1} \xrightarrow[eot_k,\, lot_k]{(s_{k-1},a_k,eot_k,lot_k,s_k)} s_k$ $(k \ge 0)$ in $\mathbf{c2et}(\mathcal{C})$. This means that $s_{in} \xrightarrow[eot_1,\, lot_1]{a_1} s_1 \ \ldots\ s_{k-1} \xrightarrow[eot_k,\, lot_k]{a_k} s_k$ in $\mathcal{C}$. Since (σ,λ) is a morphism, we may conclude that $\sigma(s_{in}) = s'_{in} \xrightarrow[eot'_1,\, lot'_1]{\lambda(a_1)} \sigma(s_1)\ \ldots\ \sigma(s_{k-1}) \xrightarrow[eot'_k,\, lot'_k]{\lambda(a_k)} \sigma(s_k)$ in $\mathbf{et2c}(\mathcal{ET}')$ for some eot'_1, …, eot'_k, lot'_1, …, $lot'_k \in \mathbf{R}$ such that $eot'_j \le eot_j$ and $lot_j \le lot'_j$ for all $1 \le j \le k$. This implies that $\sigma(s_{in}) = s'_{in} \xrightarrow[eot'_1,\, lot'_1]{e'_1} \sigma(s_1)\ \ldots\ \sigma(s_{k-1}) \xrightarrow[eot'_k,\, lot'_k]{e'_k} \sigma(s_k)$ in $\mathcal{ET}'$ for some $e'_1,\ldots,e'_k \in E'$ such that $l'(e'_j) = \lambda(a_j)$ for all $1 \le j \le k$. Moreover, it is easy to see that $g(s_{j-1},\, a_j,\, eot_j,\, lot_j,\, s_j) = e'_j$ $(1 \le j \le k)$.
- Note that $Eot'(g(s,a,eot,lot,s')) \le eot' \le eot = Eot(s,a,eot,lot,s')$ and $Lot(s,\, a,\, eot,\, lot,\, s') = lot \le lot' \le Lot'(g(s,a,eot,lot,s'))$.

Thus, (g,λ) is indeed a morphism of **TET** from $\mathbf{c2et}(\mathcal{C})$ to $\mathcal{ET}'$.

It is easy to see that $(\sigma,\lambda) = (\sigma_g,\lambda)$ and g is a unique function such that $\sigma_g = \sigma$. Furthermore, $(\mathbf{c2et}(\mathcal{C}),(1_S,1_L))$ is a reflection of $\mathcal{C}$ along $\mathbf{et2c}$. □

Now we can summarize the obtained results in order to introduce an adjunction between **TET** and **TCT**.

Theorem 2. *The functor* **et2c** *is right adjoint to* **c2et** *and this adjunction is a coreflection.*

Доказательство. The first statement follows from Proposition 5 and from the fact that for all morphisms $(\sigma,\lambda):\mathcal{C}=(S,\ s_{in},\ L,\ T,<)\to\mathcal{C}'=(S',\ s'_{in},\ L',\ T',<')$ it is true that $(1_{S'},1_{L'})\circ(\sigma,\lambda)=(\sigma,\lambda)=\mathbf{et2c}(\mathbf{c2et}(\sigma,\lambda))=\mathbf{et2c}(\mathbf{c2et}(\sigma,\lambda))\circ(1_S,1_L)$. Next, due to Proposition 5 we may conclude that the unit ψ associates each timed causal tree $\mathcal{C}=(S,\ s_{in},\ L,\ T,<)$ with the isomorphism $(1_S,1_L):\mathcal{C}\to\mathbf{et2c}(\mathbf{c2et}(\mathcal{C}))$. Hence, ψ is a natural isomorphism. □

Thus, **TCT** embeds fully and faithfully into **TET** and is equivalent to the full subcategory of **TET** consisting of those timed event trees $\mathcal{ET}$ that are isomorphic to $\mathbf{c2et}(\mathbf{et2c}(\mathcal{ET}))$.

4.3. A reflection between the categories TES and TET. This subsection is dedicated to investigation of the categories **TES** and **TET** and a relation between them. The runs of a timed event structure can be ordered in a tree. Hence, any timed event structure forms a timed event tree whose states are the runs of the timed event structure. This gives rise to a functor $\mathbf{e2et}:\mathbf{TES}\to\mathbf{TET}$.

Definition 22. *Let* $\mathcal{E}=(E,\ <,\ Con,\ L,\ l,\ Eot,\ Lot)$ *and* $\mathcal{E}'=(E',\ <',\ Con',\ L',\ l',\ Eot',\ Lot')$ *be timed event structures and* $(\mu,\lambda):\mathcal{E}\to\mathcal{E}'$ *be a morphism from* **TES**. *Define* $\mathbf{e2et}(\mathcal{E})=(S,\ \epsilon,\ E,\ Tran,\ <,\ L,\ l,\ Eot,\ Lot)$, *where* $S=\{e_1\dots e_n\in E^*\mid n\geq 0,\{e_1,\dots,e_n\}\in\mathbf{C}(\mathcal{E})$ *and for all* $1\leq i,j\leq n$ *if* $e_i<e_j$ *then* $i<j\}$ *and* $Tran=\{(e_1\dots e_n,e_{n+1},Eot(e_{n+1}),Lot(e_{n+1}),e_1\dots e_ne_{n+1})\mid e_1\dots e_n,\ e_1\dots e_ne_{n+1}\in S\}$; *and define* $\mathbf{e2et}(\mu,\lambda)=(\mu,\lambda)$.

Proposition 6. *The mapping* **e2et** *is a fully faithful functor.*

Доказательство. First, we need to show that $\mathbf{e2et}(\mathcal{E})$ is a timed event tree for all timed event structures $\mathcal{E}$. Using the definition of the sets S and $Tran$, we may easy check that $(S,\ \epsilon,\ E,\ Tran)$ is a timed synchronization tree. Note that $<\subseteq E\times E$ is a strict order, because $\mathcal{E}$ is a timed event structure. Next, we should prove that $\mathbf{e2et}(\mathcal{E})$ satisfies the requirements of Definition 17:

(i) for all $e\in E$ there exists a transition $(s,e,eot,lot,s')\in Tran$.

Clearly, $C=e\downarrow\cup\{e\}\in\mathbf{C}(\mathcal{E})$. W.l.o.g. assume that $C=\{e_1,\dots,e_n\}$ for some $n\geq 0$ such that $e_n=e$ and for all $1\leq i,j\leq n$ if $(e_i<e_j)$ then $i<j$. Define $s_i=e_1\dots e_i$ for all $1\leq i\leq n$ and $s_0=s_{in}=\epsilon$. Obviously, for all $1\leq i\leq n$, $s_i\in S$ and $(s_{i-1},e_i,Eot(e_i),Lot(e_i),s_i)\in Tran$.

(ii) if $(s,e,eot,lot,s'),(s,e,eot',lot',s'')\in Tran$, then $(s,\ e,\ eot,\ lot,\ s')=(s,\ e,\ eot',\ lot',\ s'')$.

Due to the definition of the set $Tran$, we have that $s=e_1^*\dots e_m^*$ for some $m\geq 0$, $s'=e_1^*\dots e_m^*e$, $s''=e_1^*\dots e_m^*e$ and $eot=eot'=Eot(e)$ and $lot=lot'=Lot(e)$. Hence, $(s,e,eot,lot,s')=(s,e,eot',lot',s'')$.

(iii) if (s, e, eot, lot, s'), $(u, e, eot', lot', u') \in Tran$, then there is no sequence $s_0 \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ldots s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ $(k \geq 0)$ such that $(s' = s_0 \wedge u = s_k)$ or $(u' = s_0 \wedge s = s_k)$.

Suppose that (s, e, eot, lot, s'), $(u, e, eot', lot', u') \in Tran$. By the construction of the set $Tran$, we have that $s = e_1^* \ldots e_m^*$ for some $m \geq 0$, $s' = e_1^* \ldots e_m^* e$, $u = e_1' \ldots e_k'$ for some $k \geq 0$ and $u' = e_1' \ldots e_k' e$. This means that $e \in s'$, $e \in u'$, $e \notin s$ and $e \notin u$. It is easy to see that if $s_0 \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ldots s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ $(k \geq 0)$ then s_0 is a prefix of s_k. Hence, if $s' = s_0$ then $u \neq s_k$ and if $u' = s_0$ then $s \neq s_k$.

(iv) if $e < e'$ and $(s, e', eot, lot, s') \in Tran$, then there is a sequence $s_0 \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ldots s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ $(k \geq 0)$ such that $e_1 = e$ and $s = s_k$.

Since $(s, e', eot, lot, s') \in Tran$ we have that $s = e_1^* \ldots e_m^*$ for some $m \geq 0$, $s' = e_1^* \ldots e_m^* e'$ and $\{e_1^*, \ldots, e_m^*\}$, $\{e_1^*, \ldots, e_m^*, e'\} \in \mathbf{C}(\mathcal{E})$. Hence, $e \in \{e_1^*, \ldots, e_m^*\}$. W.l.o.g. assume $e = e_j^*$ for some $1 \leq j \leq m$. Define $s_i^* = e_1^* \ldots e_i^*$ for all $1 \leq i \leq m$ and $s_{m+1}^* = e_1^* \ldots e_m^* e'$. Clearly, $s_{j-1}^* \xrightarrow[Eot(e_j),\ Lot(e_j)]{e_j = e} s_j^* \ldots s_{m-1}^* \xrightarrow[Eot(e_m),\ Lot(e_m)]{e_m} s_m^* \xrightarrow[Eot(e'),\ Lot(e')]{e'} s_{m+1}^* (m \geq 0)$.

(v) $Eot(e) \leq Lot(e)$ for all $e \in E$.

This follows from the fact that $\mathcal{E}$ is a timed event structure.

(vi) for all $(s, e, eot, lot, s_1) \in Tran$ $Eot(e) \leq eot \leq lot \leq Lot(e)$.

Clearly, for all $(s, e, eot, lot, s_1) \in Tran$ $Eot(e) = eot \leq lot = Lot(e)$.

Thus, $\mathbf{e2et}(\mathcal{E})$ is a timed event tree.

Second, we need to prove that $\mathbf{e2et}(\mu, \lambda) : \mathbf{e2et}(\mathcal{E}) \to \mathbf{e2et}(\mathcal{E}')$ is a morphism of **TET** for all morphisms $(\mu, \lambda) : \mathcal{E} \to \mathcal{E}'$ of **TES**. W.l.o.g. assume that $\mathcal{E} = (E, <, Con, L, l, Eot, Lot)$ and $\mathcal{E}' = (E', <', Con', L', l', Eot', Lot')$. Then, $\mathbf{e2et}(\mathcal{E}) = (S, \epsilon, E, Tran, <, L, l, Eot, Lot)$ and $\mathbf{e2et}(\mathcal{E}') = (S', \epsilon, E', Tran', <', L', l', Eot', Lot')$. Since $(\mu, \lambda) : \mathcal{E} \to \mathcal{E}'$ is a morphism of **TES**, we get $\mu : E \to E'$ and $\lambda : L \to L'$ are functions and $l' \circ \mu = \lambda \circ l$. Check that (μ, λ) satisfies the requirements of Definition 18.

- Let us show that $\mu(e)\downarrow \subseteq \mu(e\downarrow)$.

  Take an arbitrary $e \in E$. Obviously, $C = e\downarrow \cup \{e\} \in \mathbf{C}(\mathcal{E})$. Hence, $\mu\ C \in \mathbf{C}(\mathcal{E}')$. Since $\mu(e) \in \mu\ C$, we have that $\mu(e)\downarrow \subseteq \mu\ C = \mu(e\downarrow) \cup \{\mu(e)\}$. Because (μ, λ) is a morphism of **TES**, we have $\mu(e)\downarrow \cap \{\mu(e)\} = \varnothing$. Thus, $\mu(e)\downarrow \subseteq \mu(e\downarrow)$.

- Let $s_{in} \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ldots s_{n-1} \xrightarrow[eot_n,\ lot_n]{e_n} s_n$ for some $n \geq 0$. Due to the definition of the set $Tran$, we have that $s_{in} = \epsilon$, and for all $1 \leq i \leq n$ $s_i = e_1 \ldots e_i \in S$, $eot_i = Eot(e_i)$ and $lot_i = Lot(e_i)$. Since $s_i \in S$ $(1 \leq i \leq n)$, we get $\{e_1, \ldots, e_i\} \in \mathbf{C}(\mathcal{E})$ for all $1 \leq i \leq n$. Because (μ, λ) is a morphism of **TES**, it holds that $\{\mu(e_1), \ldots, \mu(e_i)\} \in \mathbf{C}(\mathcal{E}')$ for all $1 \leq i \leq k$. Define

$s'_i = \mu(e_1)\dots\mu(e_i)$ for all $1 \le i \le n$. Clearly, $s'_i \in S'$ $(1 \le i \le k)$ and $s'_{in} \xrightarrow[Eot'(\mu(e_1)),\ Lot'(\mu(e_1))]{\mu(e_1)} s'_1 \dots s'_{n-1} \xrightarrow[Eot'(\mu(e_n)),\ Lot'(\mu(e_n))]{\mu(e_n)} s'_n$, $Eot'(\mu(e_j)) \le Eot(e_j)$ and $Lot(e_j) \le Lot'(\mu(e_j))$ for all $1 \le j \le k$.

- Clearly, $Eot'(\mu(e)) \le Eot(e)$ and $Lot(e) \le Lot'(\mu(e))$ for all $e \in E$, since (μ, λ) is a morphism of **TES**.

This means that (μ, λ) is indeed a morphism of **TET** from $\mathbf{e2et}(\mathcal{E})$ to $\mathbf{e2et}(\mathcal{E}')$.

Third, consider an identity morphism $(1_E, 1_L) : \mathcal{E} \to \mathcal{E}$ and a pair of morphisms $(\mu, \lambda) : \mathcal{E} \to \mathcal{E}'$ and $(\mu', \lambda') : \mathcal{E}' \to \mathcal{E}''$ from **TES**. Clearly, $\mathbf{e2et}(1_E, 1_L) = (1_E, 1_L)$, and $\mathbf{e2et}((\mu', \lambda') \circ (\mu, \lambda)) = \mathbf{e2et}(\mu' \circ \mu, \lambda' \circ \lambda) = (\mu' \circ \mu, \lambda' \circ \lambda) = (\mu', \lambda') \circ (\mu, \lambda) = \mathbf{e2et}((\mu', \lambda')) \circ \mathbf{e2et}(\mu, \lambda)$. Thus, we have that **e2et** is indeed a functor.

Finally, we need to show that **e2et** is a fully faithful functor. Take an arbitrary objects $\mathcal{E}$ and $\mathcal{E}'$ of **TES**. Define a function $F_{\mathcal{E},\mathcal{E}'} : \mathbf{TES}(\mathcal{E}, \mathcal{E}') \to \mathbf{TET}(\mathbf{e2et}(\mathcal{E}), \mathbf{e2et}(\mathcal{E}'))$ such that $F_{\mathcal{E},\mathcal{E}'}(\mu, \lambda) = \mathbf{e2et}(\mu, \lambda) = (\mu, \lambda)$ for all morphisms $(\mu, \lambda) : \mathcal{E} \to \mathcal{E}'$ of **TES**. It is obvious that $F_{\mathcal{E},\mathcal{E}'}$ is a function, because **e2et** is a functor. It is easy to see that $F_{\mathcal{E},\mathcal{E}'}$ is injective, because $F_{\mathcal{E},\mathcal{E}'}(\mu, \lambda) = (\mu, \lambda)$. Hence, **e2et** is a faithful functor. Check that $F_{\mathcal{E},\mathcal{E}'}$ is a surjective function. Take an arbitrary morphism $(\mu, \lambda) : \mathbf{e2et}(\mathcal{E}) \to \mathbf{e2et}(\mathcal{E}')$ of **TET**. Since (μ, λ) is a morphism of **TET**, we may conclude that $\mu : E \to E'$ and $\lambda : L \to L'$ are functions, $l' \circ \mu = \lambda \circ l$ and $Eot'(\mu(e)) \le Eot(e)$ and $Lot(e) \le Lot'(\mu(e))$ for all $e \in E$. Let C be a configuration of $\mathcal{E}$. By the definition of the sets of states of $\mathbf{e2et}(\mathcal{E})$ and $\mathbf{e2et}(\mathcal{E}')$, we get that $\mu\ C \in \mathbf{C}(\mathcal{E}')$ and for all $e, e' \in C$ if $\mu(e) = \mu(e')$ then $e = e'$. This implies that (μ, λ) is a morphism of **TES** and $F_{\mathcal{E},\mathcal{E}'}(\mu, \lambda) = (\mu, \lambda)$. Therefore, **e2et** is a full functor. □

Note that we can transform any timed event tree into a timed event structure, defining the set of consistent events as a set of events that appear together on some branch and ignoring the tree structure. Thus we obtain a functor $\mathbf{et2e} : \mathbf{TET} \to \mathbf{TES}$.

Definition 23. *Let* $\mathcal{ET} = (S, s_{in}, E, T, <, L, l, Eot, Lot)$ *and* $\mathcal{ET}' = (S', s'_{in}, E', T', <', L', l', Eot', Lot')$ *be timed event trees. Define* $\mathbf{et2e}(\mathcal{ET})$ *as* $(E, <, Con, L, l, Eot, Lot)$*, where* Con *exactly contains all subsets* A *of the sets* $\{e_1, \dots, e_k\} \subseteq E$ $(k \ge 0)$ *such that there are states* $s_1, \dots, s_k \in S$ *with* $s_{in} \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \dots s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ *for some* $eot_1, \dots, eot_k, lot_1, \dots, lot_k \in \mathbf{R}$*. Moreover,* $\mathbf{et2e}(\mu, \lambda) = (\mu, \lambda)$*.*

Proposition 7. *The mapping* **et2e** *is a faithful functor.*

Доказательство. First, we need to show that $\mathbf{et2e}(\mathcal{ET})$ is a timed event structure for all timed event trees $\mathcal{ET}$. It follows from Definition 17 and Lemma 4.

Second, we have to prove that $\mathbf{et2e}(\mu,\lambda):\mathbf{et2e}(\mathcal{ET})\to\mathbf{et2e}(\mathcal{ET}')$ is a morphism of **TES** for all morphisms $(\mu,\lambda):\mathcal{ET}\to\mathcal{ET}'$ of **TET**. Since $(\mu,\lambda):\mathcal{ET}\to\mathcal{ET}'$ is a morphism of **TET**, we may conclude that $\mu:E\to E'$ and $\lambda:L\to L'$ are functions and $l'\circ\mu=\lambda\circ l$.

Take an arbitrary configuration C in the timed event structure $\mathbf{et2e}(\mathcal{ET})$. Check that $\mu\ C\in\mathbf{C}(\mathbf{et2e}(\mathcal{ET}'))$.

Since C is a configuration, it holds that $C\in Con$ and if $e<e'\in C$ then $e\in C$. Hence, there exist events $e_1,\ldots,e_k\in E$ such that $s_{in}\xrightarrow[eot_1,\ lot_1]{e_1}s_1\ \ldots\ s_{k-1}\xrightarrow[eot_k,\ lot_k]{e_k}s_k$ $(k\geq 0)$ in $\mathcal{ET}$ for some $s_1,\ \ldots,\ s_k\in S$ and $C\subseteq\{e_1,\ldots,e_k\}$. According to the item (iii) of Definition 18, we get that $s'_{in}\xrightarrow[eot'_1,\ lot'_1]{\mu(e_1)}s'_1\ \ldots\ s'_{k-1}\xrightarrow[eot'_k,\ lot'_k]{\mu(e_k)}s'_k$ $(k\geq 0)$ in $\mathcal{ET}'$ for some $s'_1,\ldots,s'_k\in S'$ and for some $eot'_1,\ldots,eot'_k,lot'_1,\ldots,lot'_k\in\mathbf{R}$ such that $eot'_j\leq eot_j$ and $lot_j\leq lot'_j$ for all $1\leq j\leq k$. Thus $\{\mu(e_1),\ldots,\mu(e_k)\}\in Con'$ and $\mu\ C\subseteq\{\mu(e_1),\ldots,\mu(e_k)\}$. Hence, $\mu\ C\in Con'$.

Let $e'\in\mu\ C$ and $e''<'e'$. This means that $e'=\mu(e_j)$ for some $1\leq j\leq k$ such that $e_j\in C$. Thus, $e''\in\mu(e_j)\downarrow$. According to item (i) of definition 18 we have that $\mu(e_j)\downarrow\subseteq\mu(e_j\downarrow)$. Using the fact that C is a configuration, we may conclude that $e''\in\mu(e_j)\downarrow\subseteq\mu(e_j\downarrow)\subseteq\mu\ C$. Thus, $\mu\ C$ is a configuration.

Now we need to show that $\forall e,e'\in C\ \diamond$ if $\mu(e)=\mu(e')$ then $e=e'$. Assume that it is not true. Then we have $e,e'\in C$ such that $\mu(e)=\mu(e')$ and $e\neq e'$. This implies that $e=e_j$ and $e'=e_l$ for some $1\leq j,l\leq k$. W.l.o.g. assume that $j\leq l$. Then there is a sequence $s'_{j-1}\xrightarrow[eot'_j,\ lot'_j]{\mu(e_j)}s'_j\ \ldots\ s'_{l-1}\xrightarrow[eot'_l,\ lot'_l]{\mu(e_l)=\mu(e_j)}s'_l$. This contradicts the item (iii) of Definition 17.

Note that $Eot'(\mu(e))\leq Eot(e)$ and $Lot(e)\leq Lot'(\mu(e))$ for all $e\in E$ due to the item (iv) of Definition 18.

Thus, (μ,λ) is a morphism of **TES** between $\mathbf{et2e}(\mathcal{ET})$ and $\mathbf{et2e}(\mathcal{ET}')$ by Definition 16.

Third, consider an identity morphism $(1_E,1_L):\mathcal{ET}\to\mathcal{ET}$ and a pair of morphisms $(\mu,\lambda):\mathcal{ET}\to\mathcal{ET}'$ and $(\mu',\lambda'):\mathcal{ET}'\to\mathcal{ET}''$ from **TET**. Obviously, $\mathbf{et2e}(1_E,1_L)=(1_E,1_L)$, and $\mathbf{et2e}((\mu',\lambda')\circ(\mu,\lambda))=\mathbf{et2e}(\mu'\circ\mu,\lambda'\circ\lambda)=(\mu'\circ\mu,\lambda'\circ\lambda)=(\mu',\lambda')\circ(\mu,\lambda)=\mathbf{et2e}(\mu',\lambda')\circ\mathbf{et2e}(\mu,\lambda)$. Thus, we have that **et2e** is indeed a functor.

Finally, we should prove that **et2e** is a faithful functor. Take arbitrary objects $\mathcal{ET}$ and $\mathcal{ET}'$ of **TET**. Define a function $F_{\mathcal{ET},\mathcal{ET}'}:\mathbf{TET}(\mathcal{ET},\mathcal{ET}')\to\mathbf{TES}(\mathbf{et2e}(\mathcal{ET}),\mathbf{et2e}(\mathcal{ET}'))$ such that $F_{\mathcal{ET},\mathcal{ET}'}(\mu,\lambda)=\mathbf{et2e}(\mu,\lambda)=(\mu,\lambda)$ for all morphisms $(\mu,\lambda):\mathcal{ET}\to\mathcal{ET}'$ of **TET**. It is obvious that $F_{\mathcal{ET},\mathcal{ET}'}$ is a function, because **et2e** is a functor. Clearly, $F_{\mathcal{ET},\mathcal{ET}'}$ is injective, because $F_{\mathcal{ET},\mathcal{ET}'}(\mu,\lambda)=(\mu,\lambda)$. Hence, **et2e** is a faithful functor. □

Proposition 8. *Let $\mathcal{E}=(E,\ <,\ Con,\ L,\ l,\ Eot,\ Lot)$ be a timed event structure. Then $\mathbf{e2et}(\mathcal{E})$ is a timed event tree, $(1_E,1_L):\mathbf{et2e}(\mathbf{e2et}(\mathcal{E}))\to\mathcal{E}$ is an isomorphism and the pair $(\mathbf{e2et}(\mathcal{E}),(1_E,1_L))$ is a coreflection of $\mathcal{E}$ along* **et2e**.

Доказательство. Obviously, $\mathbf{e2et}(\mathcal{E})=(S,\ \epsilon,\ E,\ Tran,\ <,\ L,\ l,\ Eot,\ Lot)$, where $S=\{e_1\ldots e_n$

$\in E^* \mid n \geq 0, \{e_1, \ldots, e_n\} \in \mathbf{C}(\mathcal{E})$ and for all $1 \leq i, j \leq n$ if $(e_i < e_j)$ then $(i < j)\}$ and $Tran = \{(e_1 \ldots e_n,\ e_{n+1},\ Eot(e_{n+1}),\ Lot(e_{n+1}),\ e_1 \ldots e_n e_{n+1}) \mid e_1 \ldots e_n,\ e_1 \ldots e_n e_{n+1} \in S\}$. Moreover, we can easily see that $\mathbf{et2e}(\mathbf{e2et}(\mathcal{E})) = (E, <, Con*, L, l, Eot, Lot)$, where $Con*$ exactly contains all subsets A of events from E such that $A \subseteq \{e_1, \ldots, e_k\}$ and $s_{in} \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ \ldots\ s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ $(k \geq 0)$ for some $s_1, \ldots, s_k \in S$ and $e_1, \ldots, e_k \in E$. It is easy to check that $Con = Con*$. This implies that $\mathbf{et2e}(\mathbf{e2et}(\mathcal{E})) = \mathcal{E}$. Hence, we have that $(1_E, 1_L) : \mathbf{et2e}(\mathbf{e2et}(\mathcal{E})) = \mathcal{E} \to \mathcal{E}$ is a morphism of **TES** and, moreover, it is an isomorphism.

Finally, show that $(\mathbf{e2et}(\mathcal{E}), (1_E, 1_L))$ is a coreflection of $\mathcal{E}$ along **et2e**. Consider a timed event tree $\mathcal{ET}' = (S', s'_{in}, E', T', <', L', l', Eot', Lot')$ and a morphism $(\mu, \lambda) : \mathbf{et2e}(\mathcal{ET}') \to \mathcal{E}$ and show that there is a unique morphism $(f, \varsigma) : \mathcal{ET}' \to \mathbf{e2et}(\mathcal{E})$ such that $(\mu, \lambda) = (1_E, 1_L) \circ \mathbf{et2e}(f, \varsigma)$. From this equation, it follows that (f, ς) must match (μ, λ), because $\mathbf{et2e}(f, \varsigma) = (f, \varsigma)$. Hence, we only need to show that (μ, λ) is a morphism of **TET** between $\mathcal{ET}'$ and $\mathbf{e2et}(\mathcal{E})$.

Clearly, $\mathbf{et2e}(\mathcal{ET}') = (E', <', Con', L', l', Eot', Lot')$, where Con' exactly contains all subsets A of events from E such that $A \subseteq \{e_1, \ldots, e_k\}$ and there are states $s_1, \ldots, s_k \in S'$ $(k \geq 0)$ such that $s_{in} \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ \ldots\ s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ $(k \geq 0)$ for some real numbers $eot_1, \ldots, eot_k$, $lot_1, \ldots, lot_k$. Because $(\mu, \lambda) : \mathbf{et2e}(\mathcal{ET}') \to \mathcal{E}$ is a morphism of **TES**, we have that $\mu : E' \to E$ and $\lambda : L' \to L$ are functions, $l \circ \mu = \lambda \circ l'$ and for all $e \in E$ it holds that $Eot(\mu(e)) \leq Eot'(e)$ and $Lot'(e) \leq Lot(\mu(e))$. Prove that (μ, λ) satisfies the other requirements from Definition 18. First, check that $\mu(e)\downarrow \subseteq \mu(e\downarrow)$. Let $e \in E'$. Since $\mathbf{et2e}(\mathcal{ET}')$ is a timed event structure, $e\downarrow \cup \{e\}$ is a configuration. Because $(\mu, \lambda) : et2e(\mathcal{ET}') \to \mathcal{E}$ is a morphism of **TES**, we have that $\mu(e\downarrow \cup \{e\})$ is a configuration too. Hence, $\mu(e)\downarrow \subseteq \mu(e\downarrow \cup \{e\}) = \mu(e\downarrow) \cup \{\mu(e)\}$. Clearly, if $e' \in \mu(e)\downarrow$ then $e' \neq \mu(e)$. Thus, $\mu(e)\downarrow \subseteq \mu(e\downarrow)$.

Next, assume that $s_{in} \xrightarrow[eot_1,\ lot_1]{e_1} s_1 \ \ldots\ s_{k-1} \xrightarrow[eot_k,\ lot_k]{e_k} s_k$ for some $k \geq 0$. Hence, $\{e_1, \ldots, e_j\} \in Con'$ for all $1 \leq j \leq k$. Moreover, for all $1 \leq j \leq k$ $\{e_1, \ldots, e_j\}$ is left-closed by Lemma 4. Thus, we get that $\{e_1, \ldots, e_j\} \in \mathbf{C}(et2e(\mathcal{ET}'))$ for all $1 \leq j \leq k$. Since $(\mu, \lambda) : et2e(\mathcal{ET}') \to \mathcal{E}$ is a morphism of **TES** it holds that $\{\mu(e_1), \ldots, \mu(e_j)\} \in \mathbf{C}(\mathcal{E})$ for all $1 \leq j \leq k$. Hence, for all $1 \leq j, l \leq k$ it holds that $\mu(e_j) < \mu(e_l) \Rightarrow j < l$. Let $s'_j = \mu(e_1), \ldots, \mu(e_j)$ for all $1 \leq j \leq k$. According to the definition of **e2et**, we have $s'_1, \ldots, s'_k \in S$ and $\epsilon \xrightarrow[Eot(\mu(e_1)),\ Lot(\mu(e_1))]{\mu(e_1)} s'_1 \ \ldots\ s'_{k-1} \xrightarrow[Eot(\mu(e_k)),\ Lot(\mu(e_k))]{\mu(e_k)} s'_k$. Moreover, according to Definition 17 and Definition 18, we have that $Eot(\mu(e_j)) \leq Eot'(e_j) \leq eot_j$ and $lot_j \leq Lot'(e_j) \leq Lot(\mu(e_j))$ for all $1 \leq j \leq k$. □

Using the results mentioned above, we can formulate the following theorem.

Theorem 3. *The functor* **et2e** *is left adjoint to the functor* **e2et** *and this adjunction is a reflection.*

Доказательство. The first part of this theorem follows from Proposition 8 and from the fact that for all morphisms $(\mu, \lambda) : \mathcal{E} \to \mathcal{E}'$ it is true that $(1_{E'}, 1_{L'}) \circ \mathbf{et2e}(\mathbf{e2et}(\mu, \lambda)) = (1_{E'}, 1_{L'}) \circ \mathbf{et2e}(\mu, \lambda) = (1_{E'}, 1_{L'}) \circ (\mu, \lambda) = (\mu, \lambda) = (\mu, \lambda) \circ (1_E, 1_L)$. Moreover, due to Proposition 8, we have that the counit η associates each timed event structure $\mathcal{E} = (E, <, Con, L, l, Eot, Lot)$ with the isomorphism $(1_E, 1_L) : \mathbf{et2e}(\mathbf{e2et}(\mathcal{E})) \to \mathcal{E}$. Hence, η is a natural isomorphism. □

Thus, **TES** embeds fully and faithfully into **TET** and is equivalent to the full subcategory of **TET** consisting of those timed event trees $\mathcal{ET}$ that are isomorphic to $\mathbf{e2et}(\mathbf{et2e}(\mathcal{ET}))$.

4.4. A coreflection between the categories TES and TST. It is a well-known fact that there exists a coreflection from the category of synchronization trees to the category of event structures. In this subsection, we try to extend this result to timed variants of the models mentioned above. Clearly, the configurations of a timed event structure can be translated to a tree. Hence, we can specify the following functor $\mathbf{e2s} : \mathbf{TES} \to \mathbf{TST}$.

Definition 24. *Let* $\mathcal{E} = (E, <, Con, L, l, Eot, Lot)$ *and* $\mathcal{E}' = (E', <', Con', L', l', Eot', Lot')$ *be timed event structures and* $(\mu, \lambda) : \mathcal{E} \to \mathcal{E}'$ *be a morphism of* **TES**. *Define* $\mathbf{e2s}(\mathcal{E}) = (S, \epsilon, L, Tran)$, *where* $S = \{e_1 \dots e_n \mid n \geq 0, \{e_1, \dots, e_n\} \in \mathbf{C}(\mathcal{E})$ *and for all* $1 \leq i, j \leq n$ $(e_i < e_j) \Rightarrow (i < j)\}$ *and* $Tran = \{(e_1 \dots e_n, l(e_{n+1}), Eot(e_{n+1}), Lot(e_{n+1}), e_1 \dots e_n e_{n+1}) \mid e_1 \dots e_n, e_1 \dots e_n e_{n+1} \in S\}$, *and* $\mathbf{e2s}(\mu, \lambda) = (\overline{\mu}, \lambda)$, *where* $\overline{\mu} : S \to S'$ *is defined as:* $\overline{\mu}(e_1 \dots e_n) = \mu(e_1) \dots \mu(e_n)$ *for all* $e_1 \dots e_n \in S$.

Proposition 9. *The mapping* **e2s** *is a faithful functor.*

Доказательство. First, by the definition of the sets S and $Tran$, we get that $\mathbf{e2s}(\mathcal{E})$ is a timed synchronization tree for all timed event structures $\mathcal{E}$.

Second, we need to prove that $\mathbf{e2s}(\mu, \lambda) : \mathbf{e2s}(\mathcal{E}) \to \mathbf{e2s}(\mathcal{E}')$ is a morphism of **TST** for all morphisms $(\mu, \lambda) : \mathcal{E} \to \mathcal{E}'$ of **TES**, where $\mathbf{e2s}(\mu, \lambda) = (\overline{\mu}, \lambda)$ and $\overline{\mu}(e_1 \dots e_n) = \mu(e_1) \dots \mu(e_n)$. Since $(\mu, \lambda) : \mathcal{E} \to \mathcal{E}'$ is a morphism of **TES**, it is easy to see that $\mu : E \to E'$ and $\overline{\mu} : S \to S'$ are functions. Check that the pair $(\overline{\mu}, \lambda)$ satisfies the requirements of Definition 18. It is obvious that $\overline{\mu}(\epsilon) = \epsilon$. Assume that $(e_1 \dots e_{k-1}, l(e_k), Eot(e_k), Lot(e_k), e_1 \dots e_{k-1} e_k) \in Tran$. This means that $s', s'' \in S'$, where $s' = \overline{\mu}(e_1 \dots e_{k-1})$ and $s'' = \overline{\mu}(e_1 \dots e_{k-1} e_k) = s' \mu(e_k)$. By construction of $Tran'$, we have $\overline{\mu}(e_1 \dots e_{k-1}) \xrightarrow[Eot'(\mu(e_k)),\ Lot'(\mu(e_k))]{l'(\mu(e_k))} \overline{\mu}(e_1 \dots e_{k-1} e_k))$. Moreover, $l' \circ \mu(e_k) = \lambda \circ l(e_k)$ and $Eot'(\mu(e_k)) \leq Eot(e_k)$ and $Lot(e_k) \leq Lot'(\mu(e_k))$, since (μ, λ) is a morphism of **TES**. Thus, $(\overline{\mu}, \lambda)$ is indeed a morphism of **TST** from $\mathbf{e2s}(\mathcal{E})$ to $\mathbf{e2s}(\mathcal{E}')$.

Third, we should contemplate an identity morphism $(1_E, 1_L) : \mathcal{E} \to \mathcal{E}$ and two morphisms $(\mu, \lambda) : \mathcal{E} \to \mathcal{E}'$ and $(\mu', \lambda') : \mathcal{E}' \to \mathcal{E}''$ from **TES**. Obviously, $\mathbf{e2s}(1_E, 1_L) = (\overline{1_E}, 1_L) = (1_S, 1_L)$, and $\mathbf{e2s}((\mu', \lambda') \circ (\mu, \lambda)) = \mathbf{e2s}(\mu' \circ \mu, \lambda' \circ \lambda) = (\overline{\mu' \circ \mu}, \lambda' \circ \lambda) = (\overline{\mu'}, \lambda') \circ (\overline{\mu}, \lambda) = \mathbf{e2s}(\mu', \lambda') \circ \mathbf{e2s}(\mu, \lambda)$. Thus, we have that **e2s** is indeed a functor.

Finally, we need to clarify that **e2s** is a faithful functor. Take an arbitrary timed event structures $\mathcal{E}$ and $\mathcal{E}'$ from **TES**. Specify a function $F_{\mathcal{E},\mathcal{E}'} : \mathbf{TES}(\mathcal{E},\mathcal{E}') \to \mathbf{TST}(\mathbf{e2s}(\mathcal{E}),\mathbf{e2s}(\mathcal{E}'))$ such that $F_{\mathcal{E},\mathcal{E}'}(\mu,\lambda) = \mathbf{e2s}(\mu,\lambda) = (\overline{\mu},\lambda)$ for all morphisms $(\mu,\lambda) : \mathcal{E} \to \mathcal{E}'$ of **TES**. Because **e2s** is a functor, we have that $F_{\mathcal{E},\mathcal{E}'}$ is a function.

Next, we need to verify that $F_{\mathcal{E},\mathcal{E}'}$ is an injective function. Take two arbitrary morphisms $(\mu_1,\lambda_1) : \mathcal{E} \to \mathcal{E}'$ and $(\mu_2,\lambda_2) : \mathcal{E} \to \mathcal{E}'$ such that $F_{\mathcal{E},\mathcal{E}'}(\mu_1,\lambda_1) = F_{\mathcal{E},\mathcal{E}'}(\mu_2,\lambda_2)$. This implies that $(\overline{\mu_1},\lambda_1) = (\overline{\mu_2},\lambda_2)$. Hence, $\lambda_1 = \lambda_2$ and $\overline{\mu_1} = \overline{\mu_2}$. Take an arbitrary event $e \in E$. Since $\mathcal{E}$ is a timed event structure, we have a configuration $\{e_1,\ldots,e_n\} = e\downarrow \cup \{e\}$ of $\mathcal{E}$ such that $e_n = e$ and for all $1 \le i,j \le n$ if $e_i < e_j$ then $i < j$. This implies that $s_i = e_1 \ldots e_i \in S$ for all $1 \le i \le n$. Since $\overline{\mu_1} = \overline{\mu_2}$, we have that $\mu_1(e_i) = \mu_2(e_i)$ for all $1 \le i \le n$. Hence, $\mu_1 = \mu_2$. Thus, $F_{\mathcal{E},\mathcal{E}'}$ is injective, i.e. **e2s** is a faithful functor. □

Next, we try to transform a timed synchronization tree $\mathcal{S}$ into some timed event structure $\mathcal{E}$, assuming that each transition of $\mathcal{S}$ represents a separate event with the same timed limits as this transition, defining the set of consistent events as a set of transitions that appear together on some branch and specifying the causal dependency relation as the hierarchy of transitions in the tree structure. Thus, we can specify a functor $\mathbf{s2e} : \mathbf{TST} \to \mathbf{TES}$.

Definition 25. *Let $\mathcal{S} = (S, s_{in}, L, T)$ and $\mathcal{S}' = (S', s'_{in}, L', T')$ be timed synchronization trees and $(\sigma,\lambda) : \mathcal{S} \to \mathcal{S}'$ be a morphism of* **TST**. *Define* $\mathbf{s2e}(\mathcal{S}) = (T, <^*, Con^*, L, l^*, Eot^*, Lot^*)$, *where* $(s,a,eot,lot,s') <^* (u,b,eot',lot',u') \iff$ *there is a sequence* $s' \xrightarrow[eot_1,\ lot_1]{a_1} s_1 \ \ldots\ s_{k-1} \xrightarrow[eot_k,\ lot_k]{a_k} s_k = u$ *for some* $k \ge 1$, $Con^* = \{A \subseteq \{t_1,\ldots,tr_k\} \mid tr_1 = (s_{in},\ a_1,\ eot_1,\ lot_1,\ s_1),\ \ldots,\ tr_k = (s_{k-1},\ a_k,\ eot_k,\ lot_k,\ s_k) \in T,\ (k \ge 0)\}$, $l^*(s,a,eot,lot,s') = a$, $Eot^*(s,a,eot,lot,s') = eot$ *and* $Lot^*(s,a,eot,lot,s') = lot$. *Moreover, define* $\mathbf{s2e}(\sigma,\lambda) = (\mu,\lambda)$, *where* $\mu(s,a,eot,lot,s') = (\sigma(s),\lambda(a),eot',lot',\sigma(s'))$ *for some* $eot',lot' \in \mathbf{R}$.

Lemma 7. *For any timed synchronization tree $\mathcal{S}$, if $C \in \mathbf{C}(\mathbf{s2e}(\mathcal{S}))$ then* $C = \{(s_{in},\ a_1,\ eot_1,\ lot_1,\ s_1),\ \ldots,\ (s_{n-1},\ a_n,\ eot_n,\ lot_n,\ s_n) \mid s_{in} \xrightarrow[eot_1,\ lot_1]{a_1} s_1 \ \ldots\ s_{n-1} \xrightarrow[eot_n,\ lot_n]{a_n} s_n\}$ *for some* $n \ge 0$.

Proposition 10. *The mapping* **s2e** *is a faithful functor.*

Доказательство. First, we need to show that $\mathbf{s2e}(\mathcal{S})$ is a timed event structure for all timed synchronization trees $\mathcal{S}$. It is easy to check that $<^*$ is a strict order and, for all $(s,a,eot,lot,s') \in T$, $Eot^*(s,a,eot,lot,s') \le Lot^*(s,a,eot,lot,s')$. Take an arbitrary $(s,a,eot,lot,s') \in T$. Now we need to verify that $(s,a,eot,lot,s')\downarrow = \{(u,b,eot',lot',u') \in T \mid (u,\ b,\ eot',\ lot',\ u') <^* (s,\ a,\ eot,\ lot,\ s')\}$ is a finite set. Since $\mathcal{S}$ is a timed synchronization tree, we can find a unique sequence $s_{in} \xrightarrow[eot_1,\ lot_1]{a_1} s_1 \ \ldots\ s_{k-1} \xrightarrow[eot_k,\ lot_k]{a_k} s_k = s$ $(k \ge 0)$. Using the definition of a timed synchronization tree, we get that $(s,\ a,\ eot,\ lot,\ s')\downarrow \subseteq \{(s_{in},\ a_1,\ eot_1,\ lot_1,\ s_1),\ \ldots,\ (s_{k-1},\ a_k,\ eot_k,\ lot_k,\ s_k)\}$. Hence, $(s,a,eot,lot,s')\downarrow$ is a finite set. Moreover, it immediately follows from the definition of

Con^* that $\forall(s,a,eot,lot,s') \in T \circ \{(s,a,eot,lot,s')\} \in Con^*$ and $Y \subseteq X \in Con^* \Rightarrow Y \in Con^*$. By the definition of the relation $<^*$, we get that for all $X \in Con^*$, if $(u,b,eot',lot',u') <^* (s,a,eot,lot,s') \in X$ then $X \cup \{(u,\ b,\ eot',\ lot',\ u')\} \in Con^*$.

Second, we have to prove that $\mathbf{s2e}(\sigma,\lambda)$ is a morphism of **TES**. Assume that $\mathbf{s2e}(\mathcal{S}) = (T, <^*, Con^*, L, l^*, Eot^*, Lot^*)$ and $\mathbf{s2e}(\mathcal{S}') = (T', <'^*, Con'^*, L', l'^*, Eot'^*, Lot'^*)$. Since $(\sigma,\lambda) : \mathcal{S} \to \mathcal{S}'$ is a morphism of **TST**, we may conclude that $\lambda : L \to L'$ and $\sigma : S \to S'$ are functions, $\sigma(s_{in}) = s'_{in}$ and for each $(s,a,eot,lot,s') \in T$ there is the only $(\sigma(s),\lambda(a),eot',lot',\sigma(s')) \in T'$, where $eot' \le eot$ and $lot \le lot'$. Hence, μ, defined as $\mu(s,a,eot,lot,s') = (\sigma(s),\ \lambda(a),\ eot',\ lot',\ \sigma(s'))$, is a function. Take an arbitrary configuration C of $\mathbf{s2e}(\mathcal{S})$ and check that $\mu\ C \in \mathbf{C}(\mathbf{s2e}(\mathcal{S}'))$. Since C is a configuration, it holds that $C \in Con^*$ and if $e <^* e' \in C$ then $e \in C$. Hence, there exist states $s_1, \ldots, s_k \in S$ $(k \ge 0)$ such that $s_{in} \xrightarrow[eot_1,\ lot_1]{a_1} s_1 \ldots s_{k-1} \xrightarrow[eot_k,\ lot_k]{a_k} s_k$ and $C \subseteq \{(s_0,a_1,eot_1,lot_1,s_1), \ldots, (s_{k-1},a_k,eot_k,lot_k,s_k)\}$. W.l.o.g. suppose $e_i = (s_{i-1},a_i,eot_i,lot_i,s_i)$ $(i = 1,\ldots,k)$, where $s_0 = s_{in}$. Because (σ,λ) is a morphism of **TST**, we get that $\sigma(s_{in}) \xrightarrow[eot'_1,\ lot'_1]{\lambda(a_1)} \sigma(s_1) \ldots \sigma(s_{k-1}) \xrightarrow[eot'_k,\ lot'_k]{\lambda(a_k)} \sigma(s_k)$ in $\mathcal{S}'$. Moreover, it is easy to see that $\mu(e_i) = (\sigma(s_{i-1}),\lambda(a_i),eot'_i,lot'_i,\sigma(s_i))$ for all $1 \le i \le k$. Thus, $\mu\ C \subseteq \{\mu(e_1), \ldots, \mu(e_k)\}$ and $\{\mu(e_1), \ldots, \mu(e_k)\} \in Con'^*$. Clearly, for all $e_i \in C$, $\mu(e_i)\downarrow = \{\mu(e_1), \ldots, \mu(e_{i-1})\}$. Hence, $\mu\ C$ is a configuration. Note that $\forall e_i,\ e_j \in C \circ$ if $\mu(e_i) = \mu(e_j)$ then $i = j$. Moreover, it is obvious that $Eot'^*(\mu(s,\ a,\ eot,\ lot,\ s')) \le Eot^*(s,\ a,\ eot,\ lot,\ s')$ and $Lot^*(s,\ a,\ eot,\ lot,\ s') \le Lot'^*(\mu(s,\ a,\ eot,\ lot,\ s'))$ for all $(s,a,eot,lot,s') \in T$, since $\mu(s,a,eot,lot,s') = (\sigma(s),\lambda(a),eot',lot',\sigma(s'))$ with $eot' \le eot$ and $lot \le lot'$. Thus, (μ,λ) is indeed a morphism of **TES**.

Third, we should contemplate an identity morphism $(1_S,1_L) : \mathcal{S} \to \mathcal{S}$ and two morphisms $(\sigma,\lambda) : \mathcal{S} \to \mathcal{S}'$ and $(\sigma',\lambda') : \mathcal{S}' \to \mathcal{S}''$ from **TST**. Obviously, $\mathbf{s2e}(1_S,1_L) = (\mu_{1_E,1_L},1_L) = (1_E,1_L)$, and $\mathbf{s2e}((\sigma',\lambda') \circ (\sigma,\lambda)) = \mathbf{s2e}(\sigma' \circ \sigma, \lambda' \circ \lambda) = (\mu_{\sigma' \circ \sigma, \lambda' \circ \lambda}, \lambda' \circ \lambda) = (\mu_{\sigma',\lambda'},\lambda') \circ (\mu_{\sigma,\lambda},\lambda)$ $= \mathbf{s2e}(\sigma',\lambda') \circ \mathbf{s2e}(\sigma,\lambda)$. Hence, **s2e** is a functor.

Finally, show that **s2e** is a fully faithful functor. Take arbitrary timed synchronization trees $\mathcal{S}$ and $\mathcal{S}'$ from **TST**. Define a mapping $F_{\mathcal{S},\mathcal{S}'} : \mathbf{TST}(\mathcal{S},\mathcal{S}') \to \mathbf{TES}(\mathbf{s2e}(\mathcal{S}),\mathbf{s2e}(\mathcal{S}'))$ such that $F_{\mathcal{S},\mathcal{S}'}(\sigma,\lambda) = \mathbf{s2e}(\sigma,\lambda) = (\mu_{\sigma,\lambda},\lambda)$ for all morphisms $(\sigma,\lambda) : \mathcal{S} \to \mathcal{S}'$ of **TST**. Because **s2e** is a functor, we have that $F_{\mathcal{S},\mathcal{S}'}$ is a function.

Check that $F_{\mathcal{S},\mathcal{S}'}$ is a bijective function. Take arbitrary morphisms $(\sigma_1,\lambda_1) : \mathcal{S} \to \mathcal{S}'$ and $(\sigma_2,\lambda_2) : \mathcal{S} \to \mathcal{S}'$ such that $F_{\mathcal{S},\mathcal{S}'}(\sigma_1,\lambda_1) = F_{\mathcal{S},\mathcal{S}'}(\sigma_2,\lambda_2)$. This means that $(\mu_{\sigma_1,\lambda_1},\lambda_1) = (\mu_{\sigma_2,\lambda_2},\lambda_2)$. Hence, $\lambda_1 = \lambda_2$ and $\mu_{\sigma_1,\lambda_1} = \mu_{\sigma_2,\lambda_2}$. Take an arbitrary state $s \in S$. Because $\mathcal{S}$ is a timed synchronization tree, we can find the only transition $(s',a,eot,lot,s) \in T$. Since $\mu_{\sigma_1,\lambda_1} = \mu_{\sigma_2,\lambda_2}$, we have $(\sigma_1(s'),\lambda_1(a),eot',lot',\sigma_1(s)) = (\sigma_2(s'),\lambda_2(a) = \lambda_1(a),eot'',lot'',\sigma_2(s))$. This implies that $\sigma_1(s) = \sigma_2(s)$. Hence, $F_{\mathcal{S},\mathcal{S}'}$ is injective, i.e. **s2e** is a faithful functor. Next, take an arbitrary morphism $(\mu,\lambda) : \mathbf{s2e}(\mathcal{S}) \to \mathbf{s2e}(\mathcal{S}')$ of **TES**. Define a function $g : S \to S'$ as follows: $g(s_{in}) = s'_{in}$

and for all $s \in S$ such that $s \neq s_{in}$, $g(s) = last(\mu(tr_s))$, where $tr_s \in T$ such that $last(tr_s) = s$ and $last$ is a function which maps each transition to it's last state, i.e. $last(u, b, eot', lot', u') = u'$ for all transitions (u, b, eot', lot', u'). Since $\mathcal{S}$ is a timed synchronization tree, there is the only transition $tr_s \in T$ with $last(tr_s) = s$. This means that g is indeed a function. Since $\mu(tr)\downarrow = \mu(tr\downarrow)$ for all $tr \in T$ and $l'^* \circ \mu = l^*$, we get that $(last(\mu(tr_s)), \lambda(a), eot', lot', last(\mu(tr_{s'}))) = \mu(tr_{s'}) \in T'$, where $tr_{s'} = (s, a, eot, lot, s')$. This implies that (g, λ) is a morphism of **TST** from $\mathcal{S}$ to $\mathcal{S}'$ and $F_{\mathcal{S},\mathcal{S}'}(g, \lambda) = (\mu, \lambda)$. Thus, **s2e** is a full functor. □

Proposition 11. *Let $\mathcal{S} = (S,\ s_{in},\ L,\ T)$ be a timed synchronization tree. Then there is an isomorphism $(\eta_S^*, 1_L) : \mathcal{S} \to \mathbf{e2s}(\mathbf{s2e}(\mathcal{S}))$ such that the pair $(\mathbf{s2e}(\mathcal{S}),\ (\eta^*, 1_L))$ is a reflection of $\mathcal{S}$ along* **e2s**.

Доказательство. Note, that $\mathbf{s2e}(\mathcal{S}) = (T, <^*, Con^*, L, l^*, Eot^*, Lot^*)$, where $<^*$, Con^*, l^*, Eot^* and Lot^* are defined as in Definition 25. Furthermore, $\mathbf{e2s}(\mathbf{s2e}(\mathcal{S})) = (S^*, \epsilon, L, Tran)$, where $S^* = \{(s_0, a_1, eot_1, lot_1, s_1) \ldots (s_{n-1}, a_n, eot_n, lot_n, s_n) \mid n \geq 0, \{(s_0, a_1, eot_1, lot_1, s_1), \ldots, (s_{n-1}, a_n, eot_n, lot_n, s_n)\} \in \mathbf{C}(\mathbf{s2e}(\mathcal{S}))$ and for all $1 \leq i, j \leq n$ $(s_{i-1}, a_i, eot_i, lot_i, s_i) <^* (s_{j-1}, a_j, eot_j, lot_j, s_j) \Rightarrow (i < j)\}$ and $Tran = \{((s_0, a_1, eot_1, lot_1, s_1) \ldots (s_{n-1}, a_n, eot_n, lot_n, s_n), l^*((s_n, a_{n+1}, eot_{n+1}, lot_{n+1}, s_{n+1})), Eot^*((s_n, a_{n+1}, eot_{n+1}, lot_{n+1}, s_{n+1})), Lot^*((s_n, a_{n+1}, eot_{n+1}, lot_{n+1}, s_{n+1})), (s_0, a_1, eot_1, lot_1, s_1) \ldots (s_{n-1}, a_n, eot_n, lot_n, s_n)\ (s_n, a_{n+1}, eot_{n+1}, lot_{n+1}, s_{n+1})) \mid (s_0, a_1, eot_1, lot_1, s_1) \ldots (s_{n-1}, a_n, eot_n, lot_n, s_n), (s_0, a_1, eot_1, lot_1, s_1) \ldots (s_{n-1}, a_n, eot_n, lot_n, s_n)\ (s_n, a_{n+1}, eot_{n+1}, lot_{n+1}, s_{n+1}) \in S^*\}$.

By Lemma 7 we may conclude that $S^* = \{(s_{in}, a_1, eot_1, lot_1, s_1) \ldots (s_{n-1}, a_n, eot_n, lot_n, s_n) \mid n \geq 0$ and $(s_{i-1}, a_i, eot_i, lot_i, s_i) \in T$ for all $1 \leq i \leq n\}$ and $Tran = \{((s_{in}, a_1, eot_1, lot_1, s_1) \ldots (s_{n-1}, a_n, eot_n, lot_n, s_n), a_{n+1}, eot_{n+1}, lot_{n+1}, (s_{in}, a_1, eot_1, lot_1, s_1) \ldots (s_{n-1}, a_n, eot_n, lot_n, s_n)\ (s_n, a_{n+1}, eot_{n+1}, lot_{n+1}, s_{n+1})) \mid (s_{i-1}, a_i, eot_i, lot_i, s_i) \in T$ for all $1 \leq i \leq n+1\}$.

Define a mapping $\eta_S^* : S \to S^*$ as follows: for all $s \in S \circ \eta_S^*(s) = (s_{in}, a_1, eot_1, lot_1, s_1) \ldots (s_{k-1}, a_k, eot_k, lot_k, s_k))$, where $s_k = s$. It is easy to see that for all $s \in S$ there is a unique sequence $s_{in} \xrightarrow[eot_1,\ lot_1]{a_1} s_1 \ldots s_{k-1} \xrightarrow[eot_k,\ lot_k]{a_k} s_k = s$ with $k \geq 0$ by Definition 12. Hence, η_S^* is a function and $\eta_S^*(s_{in}) = \epsilon$.

Define a mapping $\eta_S^{**} : S^* \to S$ as follows: for all $s^* \in S^* \circ \eta_S^{**}(s^*) = \eta_S^{**}((s_{in}, a_1, eot_1, lot_1, s_1) \ldots (s_{k-1}, a_k, eot_k, lot_k, s_k)) = s_k$ and $\eta_S^{**}(\epsilon) = s_{in}$. Clearly, η_S^{**} is a function and $\eta_S^{**} \circ \eta_S^* = 1_S$ and $\eta_S^* \circ \eta_S^{**} = 1_{S^*}$.

Now, we need to prove that $(\eta_S^*, 1_L) : \mathcal{S} \to \mathbf{e2s}(\mathbf{s2e}(\mathcal{S}))$ is a morphism of **TST**. Obviously, η_S^* and 1_L are functions and $\eta_S^*(s_{in}) = \epsilon$. Take an arbitrary $(s, a, eot, lot, s') \in T$. According to Definition 12, we have a unique sequence $s_{in} \xrightarrow[eot_1,\ lot_1]{a_1} s_1 \ldots s_{k-1} \xrightarrow[eot_k,\ lot_k]{a_k} s_k = s$. Hence, $\eta_S^*(s) = (s_{in}, a_1, eot_1, lot_1, s_1) \ldots (s_{k-1}, a_k, eot_k, lot_k, s_k)$, $\eta_S^*(s') = (s_{in}, a_1, eot_1, lot_1, s_1) \ldots (s_{k-1}, a_k, eot_k, lot_k, s_k)\ (s, a, eot, lot, s')$ and $(\eta_S^*(s), a, eot, lot, \eta_S^*(s')) \in Tran$. Thus,

$(\eta_S^*, 1_L) : \mathcal{S} \to \mathbf{e2s}(\mathbf{s2e}(\mathcal{S}))$ is really a morphism of **TST**.

Next, we need to show that $(\eta_S^{**}, 1_L) : \mathbf{e2s}(\mathbf{e2c}(\mathcal{S})) \to \mathcal{S}$ is a morphism of **TST**. Obviously, η_S^{**} is a function and $\eta_S^{**}(\epsilon) = s_{in}$. Suppose that $(t,\, a,\, eot,\, lot,\, t') \in Tran$ for some $t, t' \in S^*$. This means that $t = (s_{in},\, a_1,\, eot_1,\, lot_1,\, s_1) \ldots (s_{k-1},\, a_k,\, eot_k,\, lot_k,\, s_k)$, $t' = (s_{in},\, a_1,\, eot_1,\, lot_1,\, s_1) \ldots (s_{k-1},\, a_k,\, eot_k,\, lot_k,\, s_k)\, (s_k,\, a,\, eot,\, lot,\, s_{k+1})$ and $(s_k,\, a,\, eot,\, lot,\, s_{k+1}) \in T$. Since $\eta_S^{**}(t) = s_k$ and $\eta_S^{**}(t') = s_{k+1}$, we may conclude that $(\eta_S^{**}, 1_L) : \mathbf{e2s}(\mathbf{s2e}(\mathcal{S})) \to \mathcal{S}$ is indeed a morphism of **TST**.

Hence, $(\eta_S^*, 1_L)$ and $(\eta_S^{**}, 1_L)$ are morphisms of **TST** and $(\eta_S^{**}, 1_L) \circ (\eta_S^*, 1_L) = (1_S, 1_L)$ and $(\eta_S^*, 1_L) \circ (\eta_S^{**}, 1_L) = (1_{S^*}, 1_L)$. Thus, $(\eta_S^*, 1_L)$ is an isomorphism.

Finally, check that $(\mathbf{s2c}(\mathcal{S}), (\eta_S^*, 1_L))$ is a reflection of $\mathcal{S}$ along **e2s**, i.e. whenever $\mathcal{E}'$ is a timed event structure and $(\sigma, \lambda) : \mathcal{S} \to \mathbf{e2s}(\mathcal{E}')$ is a morphism of **TST**, there exists a unique morphism $(g, \lambda') : \mathbf{s2e}(\mathcal{S}) \to \mathcal{E}'$ such that $(\sigma, \lambda) = \mathbf{e2s}(g, \lambda') \circ (\eta_S^*, 1_L)$. Since $\mathbf{e2s}(g, \lambda') = (\overline{g}, \lambda')$, we may conclude that λ' must be equal to λ and $\overline{g} \circ \eta_S^*$ must match σ.

Take an arbitrary timed event structure $\mathcal{E}' = (E', <', Con', L', l', Eot', Lot')$ and an arbitrary morphism $(\sigma, \lambda) : \mathcal{S} \to \mathbf{e2s}(\mathcal{E}')$ of **TST**. It is obvious that $\mathbf{e2s}(\mathcal{E}') = (S', \epsilon, L', Tran')$, where $S' = \{e_1 \ldots e_n \mid n \geq 0,\ \{e_1, \ldots, e_n\} \in \mathbf{C}(\mathcal{E}) \text{ and for all } 1 \leq i, j \leq n\ (e_i < e_j) \Rightarrow (i < j)\}$ and $Tran' = \{(e_1 \ldots e_n,\, l'(e_{n+1}),\, Eot'(e_{n+1}),\, Lot'(e_{n+1}),\, e_1 \ldots e_n e_{n+1}) \mid e_1 \ldots e_n,\, e_1 \ldots e_n e_{n+1} \in S'\}$. By definition of morphism of **TST**, it holds that $\sigma : S \to S'$ and $\lambda : L \to L'$ are functions, $\sigma(s_{in}) = \epsilon$ and for all $(s,\, a,\, eot,\, lot,\, s') \in T$ there exist $eot', lot' \in \mathbf{R}$ such that $eot' \leq eot$, $lot \leq lot'$ and $(\sigma(s), \lambda(a), eot', lot', \sigma(s')) \in Tran'$.

Clearly, $\overline{g} \circ \eta_S^* = \sigma \iff$ for all $(s, e, eot, lot, s') \in T$ $g(s, e, eot, lot, s') = e_k$, where $\sigma(s') = e_1 \ldots e_k$ for some $k \geq 0$. We should only show that $(g, \lambda) : \mathbf{s2e}(\mathcal{S}) \to \mathcal{E}'$ is a morphism of **TES**, where $g(s, e, eot, lot, s') = e_k$ with $\sigma(s') = e_1 \ldots e_k$ for some $k \geq 0$. Note that g is a function, because σ is a function. Moreover, for all $(s, a, eot, lot, s') \in T$ there exist $eot', lot' \in \mathbf{R}$ such that $eot' \leq eot$, $lot \leq lot'$ and $(\sigma(s), \lambda(a), eot', lot', \sigma(s')) \in Tran'$. By the definition of $\mathbf{e2s}(\mathcal{E}')$, we have $\sigma(s') = \sigma(s)\ e_k$ and $l'(e_k) = \lambda(a)$. This implies that $l' \circ g(s, a, eot, lot, s') = l'(e_k) = \lambda(a) = \lambda \circ l^*(s, a, eot, lot, s')$ for all $(s, a, eot, lot, s') \in T$.

Assume that $C \in \mathbf{C}(\mathbf{s2e}(\mathcal{S}))$. By Lemma 7 we have that $C = \{(s_{in} = s_0,\, a_1,\, eot_1,\, lot_1,\, s_1), \ldots, (s_{n-1},\, a_n,\, eot_n,\, lot_n,\, s_n) \mid (s_{j-1},\, a_j,\, eot_j,\, lot_j,\, s_j) \in T \text{ for all } 1 \leq j \leq n\}$. It is easy to see that $\sigma(s_0) = \epsilon$, $\sigma(s_1) = e'_1, \ldots, \sigma(s_n) = e'_1\ \ldots\ e'_n$ for some $e'_1,\ \ldots,\ e'_n \in E'$. Hence, $g\ C = \{g(s_{in} = s_0,\, a_1,\, eot_1,\, lot_1,\, s_1), \ldots, g(s_{n-1},\, a_n,\, eot_n,\, lot_n,\, s_n) \mid (s_{j-1},\, a_j,\, eot_j,\, lot_j,\, s_j) \in T \text{ for all } 1 \leq j \leq n\} = \{e'_1, \ldots, e'_n \mid \sigma(s_n) = e'_1\ \ldots\ e'_n\}$. Thus, $g\ C \in \mathbf{C}(\mathcal{E}')$. Next, consider two transitions $(s_{j-1},\, a_j,\, eot_j,\, lot_j,\, s_j)$ and $(s_{i-1},\, a_i,\, eot_i,\, lot_i,\, s_i)$ from C. If $g(s_{j-1},\, a_j,\, eot_j,\, lot_j,\, s_j) = g(s_{i-1},\, a_i,\, eot_i,\, lot_i,\, s_i)$ then $e'_i = e'_j$. Hence, $i = j$.

Furthermore, for all $(s_{i-1},\, a_i,\, eot_i,\, lot_i,\, s_i) \in C$ it holds that $Eot'(g(s_{i-1},\, a_i,\, eot_i,\, lot_i,\, s_i)) = Eot'(e'_i) \leq eot_i = Eot^*(s_{i-1},\, a_i,\, eot_i,\, lot_i,\, s_i)$ and $Lot^*(s_{i-1},\, a_i,\, eot_i,\, lot_i,\, s_i) = lot_i \leq Lot'(e'_i) =$

$Lot'(g(s_{i-1},\ a_i,\ eot_i,\ lot_i,\ s_i))$. Thus, (g, λ) is indeed a morphism of **TES** between $\mathbf{s2e}(\mathcal{S})$ and $\mathcal{E}'$. Therefore, $(\mathbf{s2e}(\mathcal{S}), (\eta^*, 1_L))$ is a reflection of $\mathcal{S}$ along **e2s**. □

As a result, the following statement is true.

Theorem 4. *The functor* **e2s** *is right adjoint to* **s2e** *and the adjunction is a coreflection.*

Доказательство. The first statement follows from Proposition 11 and from the fact that for all morphisms $(\sigma, \lambda) : \mathcal{S} = (S,\ s_{in},\ L,\ T) \to \mathcal{S}' = (S',\ s'_{in},\ L',\ T')$ it is true that $(\eta^*_{S'}, 1_{L'}) \circ (\sigma, \lambda) = \mathbf{e2s}(\mathbf{s2e}(\sigma, \lambda)) \circ (\eta^*_S, 1_L)$. Next, due to Lemma 11 we may conclude that the unit ψ associates each timed synchronization tree $\mathcal{S} = (S,\ s_{in},\ L,\ T)$ with the isomorphism $(\eta^*_S, 1_L) : \mathcal{S} \to \mathbf{e2s}(\mathbf{s2e}(\mathcal{S}))$. Hence, ψ is a natural isomorphism. □

Thus, **TST** embeds fully and faithfully into **TES** and is equivalent to the full subcategory of **TES** consisting of those timed event structures $\mathcal{E}$ that are isomorphic to $\mathbf{s2e}(\mathbf{e2s}(\mathcal{E}))$.

4.5. Summary. The following diagram summarizes the functors which relate the models under consideration. Here the hooks represent embeddings and the small triangles between arrows indicate the direction of left adjoints.

$$\begin{array}{ccc} \mathbf{TCT} & \overset{\hookrightarrow}{\underset{\leftarrow}{\triangleright}} & \mathbf{TET} \\ \uparrow\!\triangle\!\downarrow & & \uparrow\!\triangledown\!\downarrow \\ \mathbf{TST} & \overset{\hookrightarrow}{\underset{\leftarrow}{\triangleright}} & \mathbf{TES} \end{array}$$

The diagram can be seen as a decomposition of the coreflection from **TST** to **TES** into three consecutive adjunctions. Moreover, it is clear that the embeddings and left adjoints commute. Thus we have derived a composed adjunction between timed causal trees and timed event structures. It is not a coreflection, but it is induced by a coreflection and a reflection via a larger category, **TET**. The object component of the right adjoint of this adjunction amounts to the following transformation: it 'linearizes' a timed event structure into a timed causal tree by forgetting about events.

5. Conclusion. In this paper we established some relations between the timed extension of the well-known concurrent models. In particular, we showed that:

- The category of timed synchronization trees embeds fully and faithfully into the category of timed event structures and into the category of timed causal trees.

- There is an adjunction between the category of timed causal trees and the category of timed event structures. This adjunction is represented as the composition of a coreflection from the category of timed causal trees to the category of timed event trees and a reflection from the category of timed event trees to the category of timed event structures.

Thus, as in the case of timeless models, timed causal trees are more trivial than timed event structures because they apply causality without the notion of an event and, at the same time, are more expressive than the latter, because their possible runs can be defined in terms of a tree without restrictions, but the set of possible runs of any event structure must be closed under the shuffling of concurrent transitions.

References

1. Borceux F. Handbook of Categorical Algebra. Vol. 2, 3. Encyclopedia of Mathematics and its Applications. Vol. 51, 52. Cambridge University Press. 1994.
2. Fröschle S., Lasota S. Causality versus true-concurrency // Theoretical Computer Science. 2007. Vol. 386 (3) P. 169–187.
3. Joyal A., Nielsen M., Winskel G. Bisimulation from open maps // Information and Computation. 1996. Vol. 127(2). P. 164–185.
4. Nielsen M., Winskel G. Petri nets and bisimulation // Theoretical Computer Science. 1996. Vol. 153 (1-2). P. 211—244.
5. Winskel G., Nielsen M. Models for concurrency. Handbook of logic in computer science. Vol. 4. Oxford Univ. Press. New York. 1995. P. 1–148.

УДК: 519.681.3, 519.681.2

Название: Сравнение причинной зависимости и семантики истинного параллелизма в контексте временных моделей

Автор(ы):

Грибовская Н.С. (Институт систем информатики СО РАН)

Аннотация: Цель данной работы — установить взаимосвязи между различными параллельными моделями реального времени. Для достижения данной цели мы определили категорию временных причинных деревьев и исследовали, какое место занимает эта категория среди других категорий временных моделей. В частности, мы установили существование сопряженных функторов между категорией временных причинных деревьев и категорией временных структур событий, используя для этого более выразительную модель временных деревьев событий. Тем самым мы показали, что временные причинные деревья проще временных структур событий в том, что они отражают только один аспект семантики истинного параллелизма, а именно причинную зависимость, и не используют понятие события для задания отношения причинной зависимости. С другой стороны, модель временных причинных деревьев более выразительна, чем модель временных структур событий по следующей причине: для нее множество всех возможных последовательностей выполнения может быть определено в терминах дерева без каких-либо ограничений,

а множество всевозможных последовательностей выполнения для временной структуры событий должно быть замкнутыми относительно операции перестановки параллельных переходов.

Ключевые слова: модели реального времени, истинный параллелизм, причинная зависимость, отношения, унификация, теория категорий

UDK: 004.02

Title: Spin for puzzles: Using Spin for solving the Japanese river puzzle and the Square-1 cube

Author(s): Evgeny V. Bodin (A.P. Ershov Institute of Informatics Systems),

Abstract: This paper describes how the SPIN model checker can be applied to solving puzzles, such as riverIQGame (an advanced "wolf, goat and cabbage" puzzle) or "Irregular IQ Cube" (also known as 'Square-1').

Keywords: SPIN, verification

1. Introduction.

Recently, formal verification of software and hardware systems becomes important for those systems whose reliabilty is crucial, for example, when the incorrectly used hardware may become dangerous to people (like medical equipment) or may be lost forever (like a spacecraft).

In the future, the need for verification specialists may increase, so it is important to prepare them now and to make young people aware of verification tools.

A possible way to popularize something is to solve puzzles with it.

The goal of the paper is to demonstrate how puzzles (non-trivial for a human being) can be solved by the SPIN verification tool [4]. Solving such puzzles can be useful for teaching students the modern automated verification tools based on the model-checking method.

Similar logical problems, like the famous "Farmer, Wolf, Goat and Cabbage problem"[1], are often used. But this problem is so well known that it becomes boring. Moreover, (almost) everyone already knows the answer. Another possible case study could be a search for fake coins among valid ones [5].

Some time ago, an advanced "Farmer, Wolf, Goat and Cabbage problem" puzzle appeared in the Internet[7], usually referred to as "Japanese river puzzle"[1]. The goal is to help eight people (a policeperson and a criminal, and a family consisting of a mother, a father and four children (two daughters and two sons)) cross the river using one small boat. The limiting rules are:

- the boat can hold at most two people (and at least one, since the empty boat cannot move);
- the father cannot stay with a daughter if the mother is not there;
- the mother cannot stay with a son if the father is not there;
- the criminal cannot stay with any person if the policeperson is not there;

[1]The strange thing is that the Flash game introduction is in Chinese.

- only the father, the mother and the policeperson know how to operate the boat.

A less known puzzle is the Irregular IQ Cube[11](see Fig. 1).[2]

In this paper, the above mentioned puzzles are solved using the SPIN model checker. SPIN (**S**imple **P**romela **In**terpreter) is a verification tool for parallel and distributed systems described in Promela (**Pro**cess/**Pro**tocol **Me**ta **La**nguage). As the SPIN website[9] says,

> Spin is a popular open-source software tool, used by thousands of people worldwide, that can be used for the formal verification of distributed software systems. The tool was developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments in the field. In April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM.

2. Japanese river puzzle specified in Promela.

First, use the Promela feature *mtype* for naming the moving entities in the puzzle:

```
mtype = {Cop, Criminal, Mom, Dad, Girl, Boy, Boat};
```

It assigns the values from 1 to 7 to these names.

Considering the original bank of the river as left, let the configuration be an array indexed with the entities, where the values represent the numbers of the corresponding entity at the right bank[3]:

```
int r[8];
```

It makes an array with indexes from 0..7, so the r[0] item is not used.

Obviously, the target condition is when the number of entities at the right bank is 1 or 2 for all of them:

```
#define DONE (r[Cop] == 1 && r[Criminal] == 1 && \
              r[Mom] == 1 && r[Dad] == 1 && \
              r[Girl] == 2 && r[Boy] == 2 && r[Boat] == 1)
```

The backslashes at the ends of lines allow a long line to be broken into several shorter ones.

Checking if a child is at the same bank with someone seems a little tricky, but it is easily seen from the following: e.g., when both boys are away from their mother, the values are:

[2]Strictly speaking, that thing is not a **cube** since not all its faces are squares.

[3]Choosing the right one is based on the fact that the Promela arrays are implicitly filled with zeroes at the beginning of the specification.

- either r[Mom] == 0 and r[Boy] = 2 (mother at left, both boys at right)
- or r[Mom] == 1 and r[Boy] = 0 (mother at right, both boys at left).

Using these definitions, let us describe the "unsafe" configuration:

```
#define NotWith(children, with) (r[children] == 2*(1-r[with]))
#define With(children, with)    (r[children] != 2*(1-r[with]))

#define CriminalUnsafe (r[Criminal] != r[Cop] && \
        (r[Criminal] == r[Mom] || r[Criminal] == r[Dad] || \
        With(Boy,Criminal) || With(Girl,Criminal) ))
#define BoysUnsafe ( With(Boy,Mom) && r[Mom]!=r[Dad] )
#define GirlsUnsafe ( With(Girl,Dad) && r[Mom]!=r[Dad] )
```

Since only three persons are allowed to drive the boat, let there be a *(driver, passenger)* pair (with a possible dummy passenger when the 'driver' goes alone). To avoid unnecessary duplication when the same pair is exchanged, let us order the possible drivers alphabetically: (Cop, Dad, Mom).

Here is the main part of the specification:

```
do
  /* move Cop (with anyone or alone) */
  :: r[Cop] == r[Boat] -> driver = Cop;
    /* Choose a 'random' passenger if it is here */
    if
        :: r[Criminal] == r[Boat] -> passenger = Criminal
        :: r[Mom] == r[Boat] -> passenger = Mom
        :: r[Dad] == r[Boat] -> passenger = Dad
        :: With(Boy,Cop) -> passenger = Boy
        :: With(Girl,Cop) -> passenger = Girl
        :: true -> passenger = 0 /* no passenger at all */
    fi;
    move(driver, passenger);
  /* move Dad (with a Boy or with Mom or alone) */
  :: r[Dad] == r[Boat]  -> driver = Dad;
    if
        :: r[Mom] == r[Boat] -> passenger = Mom
        :: With(Boy,Dad) -> passenger = Boy
```

```
        :: true -> passenger = 0
    fi;
    move(driver, passenger);
  /* move Mom (with a Girl or alone) */
  :: r[Mom] == r[Boat]  -> driver = Mom;
    if
        :: With(Girl,Mom) -> passenger = Girl
        :: true -> passenger = 0
    fi;
    move(driver, passenger);
   :: DONE -> printf("SOLVED\n"); assert(0); break;
   :: else -> printf("WHAT?!\n"); assert(0); break; /* Should never happen! */
od;
```

3. Applying SPIN to the Japanese river puzzle.

The resulting PROMELA specification[4] contains a 'do' loop whose parts (starting with '::') are executed non-deterministically (when several options are possible, they are chosen non-deterministically so that they are all searched when looking for a shortest path). When verifying, SPIN finds where the claimed assertions are violated. In this specification, there are two "assert(0);" statements that make SPIN 'think' it is an error. One of them (after the "DONE" condition) is really not an error, but the target. The other (marked with "else", which happens only when all other conditions are not met) would mean that the specification is wrong (or the puzzle cannot be solved). When one of them is reached, the verifier reports that fact and writes the 'trail', i.e. a file in a special format with a sequence of configurations that leads to the 'problem' configuration. When running with the '-t' option, SPIN uses this file to guide the execution of the specification, instead of choosing it non-deterministically.

There are two main methods for running Spin: command-line and GUI-based. The latter (using iSpin [10], jSpin or outdated XSpin) frees the user from the necessity to remember all command-line options. In this case, it is sufficient to make sure that the options "assertion violations" and "breadth-first search" are used.

Here is how the verification (using the command-line Spin) was made.

- `spin -a river_game1.pml`
  Make the source files for verification

- `gcc pan.c -DBFS -DREACH -DSAFETY -o file.pml.exe`

[4]Its complete text is given in the Appendix.

Compile the verifier[5].

- `file.pml.exe -E -n -i >__file.pml.exe.out_shortest__`
  Find a[6] shortest path to the final configuration.

- `spin -t river_game1.pml > __file1.pml.trailed__`
  Get a 'human-readable' representation of the path.

Results of running SPIN

The result of running `spin -t river_game1.pml` contains the moves that solve the river puzzle:

```
      Cop with Criminal go there.
      Cop goes back alone.
      Cop with Boy go there.
      Cop with Criminal go back.
      Dad with Boy go there.
      Dad goes back alone.
      Dad with Mom go there.
      Mom goes back alone.
      Cop with Criminal go there.
      Dad goes back alone.
      Dad with Mom go there.
      Mom goes back alone.
      Mom with Girl go there.
      Cop with Criminal go back.
      Cop with Girl go there.
      Cop goes back alone.
      Cop with Criminal go there.
      SOLVED
spin: river_game1.pml:108, Error: assertion violated
spin: text of failed assertion: assert(0)
spin: trail ends after 245 steps
#processes: 1
prev_dr = Cop
prev_pass = Criminal
```

[5]The `-DBFS` option makes the verifier use the breadth-first search method.

[6]There may be several shortest paths.

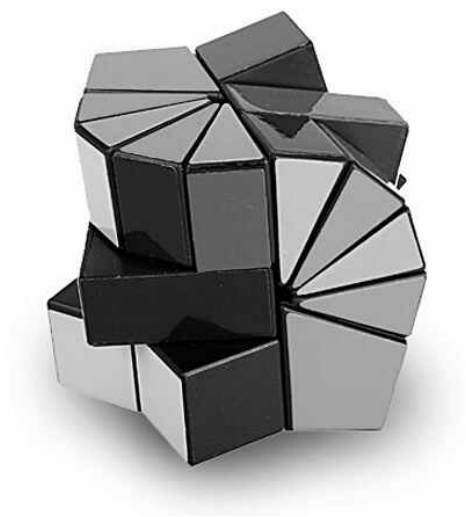


Figure 1: Irregular IQ Cube

```
r[0] = 0
r[1] = 1
r[2] = 2
r[3] = 2
r[4] = 1
r[5] = 1
r[6] = 1
r[7] = 1
245: proc  0 (:init:) river_game1.pml:111 (state 187) <valid end state>
1 process created
```

4. Irregular cube: a closer look.

After several rotations, the author failed to find his way back to the original cube configuration. Moreover, not only to its **original**, but even to **a cube**[7] configuration.

Of course, since this puzzle is not so widespread as the Rubik's cube, no complete instruction was found either. So, after some futile efforts I started to think of something that could help me.

I first met this puzzle at the think-geek site [11] where it was named 'Irregular IQ Cube', without any reference to its other names, so until recently I was not aware of the other names.[8]

[7]More or less, yes. Let us forget about it from now on and call it a cube anyway.

[8]Strangely enough, the above reference has not been valid for some time now, but it still can be found in

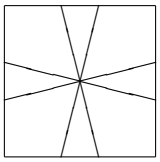


Figure 2: The target top face (the bottom face looks the same)

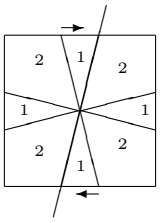


Figure 3: The target face, numbers and rotation

The first goal to reach was the "cube" form pretending that the colours are insignificant.[9]

4.1. Data representation.

The puzzle configuration can be represented as a pair of two faces: top and bottom. In the target configuration, each face is a square that consists of 8 parts: 4 triangles and 4 'deformed diamonds' (or 'kites') (see Fig. 2). Let us denote the triangles and the kites by '1's and '2's, respectively (see Fig. 3). It has some meaning: the angle of a triangle is exactly half of the kite's[10]. The sum of each face is $4 * 2 + 4 * 1 = 12$.[11]

Let us take a closer look: in order to be able to turn the halves of the cube, it must be possible to split each face by a straight line, so each face must be a pair of two halves, with the sum of the values of each half being 6. It makes sense not to consider those configurations of the faces that do not allow the cube's halves to turn, since (a) such configuration is not a target configuration and (b) to reach a target configuration, a move to a 'good' (where the 'halves-turn' is possible) configuration is necessary.

Summarizing all the above, let us make a more formal description.

Configurations.

Cube configuration: a pair of faces.

Cube face: a pair of (good) half-faces.

Half-face: a list of several 1s and 2s with their sum being 6.

Target configuration: ((2, 1, 2, 1),(2, 1, 2, 1)), ((2, 1, 2, 1), (2, 1, 2, 1)) or (2121, 2121, 2121, 2121) for short.

Kaboodle [8].

[9] In fact, they have already become less distinguishable in the process.

[10] triangle's angle is 30°, kite's angle is 60°.

[11] That is, 12 times 30°.

Possible half-faces: 111111, 21111, 12111, 11211, 11121, 11112, 2211, 2121, 2112, 1221, 1212, 1122, 222.

Possible moves.

Face rotation. A circular clockwise shift of parts belonging to one face[12], so that the resulting face is also 'good'.

Halves rotation. Rotation of the halves so that one half of the top face is interchanged with one half of the bottom face.

Full cube rotation. The top face becomes the bottom face and vice versa[13].

5. Applying SPIN to the Irregular Cube.

To make things easier, a Perl script was written to solve a half of the problem. The script creates all possible moves that lead to a different good configuration[14].

The resulting PROMELA specification[15] consists of one process `solve()` with a lengthy 'do' loop whose parts (starting with '`::`') are executed non-deterministically (when several options are possible, they are chosen non-deterministically, so that they are all searched when looking for a shortest path). The 'MSC: ' part in the `printf` statements is used to make the messages appear in the 'Message Sequence Chart' that SPIN can create in the 'simulation mode'[16].

The verification (using the command-line Spin) was made similarly to the previous example.

- `perl make_tran1_1.pl >nul`
  Create the Promela file `file.pml`.
- `spin -a file1.pml`
  Make the source files for verification.
- `gcc -DBFS -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan.exe pan.c`
  Compile the verifier.

[12]The rotation is measured in 1s (the 30° angles), so that a half-face '222' cannot be rotated by an odd number.

[13]Let us rotate only the top face and use the full cube rotation if needed. It is the only reason to introduce this kind of rotation.

[14]So that moves like $(222, 222) \xrightarrow{Rotate\ 2, 4\ or\ 6} (222, 222)$ are skipped because they lead to the same configuration, while $(222, 222) \xrightarrow{Rotate\ 1, 3\ or\ 5}$ ... are simply invalid.

[15]A fragment of it is given in the Appendix.

[16]In this case, when executing the 'trail', the sequence of configurations leads to the target configuration, where the formula claimed to be true appeared false. In fact, the '`assert(0)`' is always false, so it is just a way to pretend that this location in the specification is unreachable to have verifier complain when it finds it reachable.

- `pan.exe -m10000  -E -n >__pan.exe.out__`
  Find a shortest path to the final configuration.

- `spin -t file1.pml >__file1.pml.trailed__`
  Get a 'human-readable' representation of the path.

- `***`
  Handle the 'real-life' cube according to the path.

Human-readable trail path.

The following path is generated by running `spin -t file1.pml`.

```
      MSC: Initial state: m1221 m21111, m1221 m222
      MSC: Rotate 6 -> m21111 m1221, m1221 m222
      MSC: Rotate LEFT side -> m21111 m222, m1221 m1221
      MSC: Rotate 2 -> m2211 m1122, m1221 m1221
      MSC: Rotate LEFT side -> m2211 m1221, m1221 m1122
      MSC: Rotate 1 -> m1221 m1122, m1221 m1122
      MSC: Rotate 6 -> m1122 m1221, m1221 m1122
      MSC: Rotate LEFT side -> m1122 m1122, m1221 m1221
      MSC: Rotate 2 -> m2112 m2112, m1221 m1221
      MSC: Rotate LEFT side -> m2112 m1221, m1221 m2112
      MSC: Rotate 3 -> m2121 m1212, m1221 m2112
      MSC: Rotate LEFT side -> m2121 m2112, m1221 m1212
      MSC: Rotate RIGHT side -> m1221 m2112, m2121 m1212
      MSC: Rotate 3 -> m1212 m2121, m2121 m1212
      MSC: Rotate LEFT side -> m1212 m1212, m2121 m2121
      MSC: Rotate 2 -> m2121 m2121, m2121 m2121
      MSC: SOLVED
spin: file1.pml:1899, Error: assertion violated
spin: text of failed assertion: assert(0)
spin: trail ends after 26 steps
#processes: 1
t1 = m2121
t2 = m2121
b1 = m2121
b2 = m2121
 26: proc  0 (solve) file1.pml:1902 (state 2365) <valid end state>
1 process created
```

6. Conclusion.

This paper shows how to solve the "Japanese river puzzle" as well as the 'Irregular IQ Cube' [11].

In its preliminary version [3], the "topic of the 'future research" was specified as "to solve the cube completely", so that all cube faces have one color each. But all attempts to extend the approach 'failed miserably': verification of the Promela specifications generated by 'improved' Perl scripts (that took into account the colours of the parts) did not complete within a reasonable amount of resources (such as a 24-hour run on a machine with 12 GiB of RAM (all available RAM and swap space was consumed)). Possibly, this 'straight-forward approach' is a dead end.

At the same time, while trying to solve it, I discovered that the puzzle had a rather long history, as well as (more popular) alternative names with complete instructions (for example, at the Jaap Scherphuis' site [6][17]).

With those instructions, the puzzle can be solved without any computer. Some say that people with strong 'spacial thinking' can do it even 'intuitively', without any predefined rules.

There are people who also try to apply formal methods to puzzles or games like finding fake coins [5] or solving the checkers game [2].

References

1. Karpov, Yu. G.: *Model checking. Verification of parallel and distributed program systems*, BHV-Petersburg, 2010 (In Russian)
2. Baldamus, M., Schneider, K., Wenz, M. and Ziller, R.: Can American Checkers be Solved by Means of Symbolic Model Checking? *Electronic Notes in Theoretical Computer Science* Vol. 43, 2001, P. 15-33 Formal Methods Elsewhere (a Satellite Workshop of FORTE-PSTV-2000)
3. Bodin, E. V.: Puzzles and Spin: Irregular SPIN cube, *Program Understanding (Workshop at the 8th Ershov Informatics Conference)*, P. 4–9 Novosibirsk, 2011.
4. Holzmann, G. J.: *The Spin model checker — primer and reference manual*, Addison-Wesley, 2003.
5. Shilov, N. V. and Yi, K.: How to find a coin: propositional program logics made easy. *Current Trends in Theoretical Computer Science*, World Scientific. Vol. 2. 2004. P. 181-214.

[17] The 'Square-1' page also has a DOS/Windows program that solves an arbitrary cube using as much as 64 MiB of RAM

6. Jaap Scherphuis: (Back to) Square One / Cube 21 (online) - Jaap's Puzzle Page,
URL: http://www.jaapsch.net/puzzles/square1.htm.
7. Japanese river puzzle as a Flash game,
URL: http://freeweb.siol.net/danej/riverIQGame.swf.
8. Kaboodle site (online),
URL: http://www.kaboodle.com/reviews/thinkgeek-irregular-iq-cube.
9. Spin model checker site (online),
URL: http://spinroot.com/spin/whatispin.html.
10. Spin verifier's roadmap: Using iSpin,
URL: http://spinroot.com/spin/Man/GettingStarted.html.
11. ThinkGeek site (online),
URL: http://www.thinkgeek.com/geektoys/games/9766/.
12. Wikipedia page on Square-1 cube (online),
URL: http://en.wikipedia.org/wiki/Square_One_(puzzle).

7. Appendix 1. The resulting PROMELA specification for the Cube.

```
/* Cube Solve */

mtype= {m111111, m21111, m12111, m11211, m11121, m11112,
        m2211, m2121, m2112, m1221, m1212, m1122, m222 };
#define T1 m1221
#define T2 m21111
#define B1 m1221
#define B2 m222
#define FINAL m2121

/* Initial configuration */
mtype t1 =T1, t2=T2, b1 =B1, b2 =B2;

active proctype solve() {
  mtype tmp =0;

  printf("MSC: Initial state: %e %e, %e %e\n", t1, t2, b1, b2 );

  do
  /* Rotate */
  :: t1==m111111 && t2==m2211 -> d_step{
     t1=m21111 ; t2=m11112;
     printf("MSC: Rotate 4 -> %e %e, %e %e\n", t1, t2, b1, b2);
```

```
    }
  :: t1==m111111 && t2==m2211 -> d_step{
    t1=m2211 ; t2=m111111;
    printf("MSC: Rotate 6 -> %e %e, %e %e\n", t1, t2, b1, b2);
    }
```

. . . (Many, many similar statements)

```
  :: t1==m222 && t2==m1122 -> d_step{
    t1=m1122 ; t2=m222;
   printf("MSC: Rotate 6 -> %e %e, %e %e\n", t1, t2, b1, b2);
    }

  /* Switch halves */
  :: /* true -> */ d_step{
       tmp = t2; t2 = b2; b2 = tmp; tmp=0;
   printf("MSC: Rotate LEFT side -> %e %e, %e %e\n", t1, t2, b1, b2);
       }
  :: /* true -> */ d_step{
       tmp = t1; t1 = b1; b1 = tmp; tmp=0;
   printf("MSC: Rotate RIGHT side -> %e %e, %e %e\n", t1, t2, b1, b2);
       }
  /* Full Cube Move */
  :: true -> d_step{
       tmp = t1; t1 = b1; b1 = tmp;
       tmp = t2; t2 = b2; b2 = tmp;
       tmp=0;
   printf("MSC: Full Cube Move -> %e %e, %e %e\n", t1, t2, b1, b2);
       }
  /* Pretend that the final state is not reachable
    to have SPIN find a counter-example  */
  :: t1==FINAL && t2==FINAL && b1==FINAL && b2==FINAL ->
       printf("MSC: SOLVED\n"); assert(0); break;
  :: else -> printf("MSC: WHAT?");
       assert(0); break; /* Should never happen! */
  od;
} /* solve() */
```

8. Appendix 2. The complete PROMELA specification for the Japanese river puzzle.

```
/* Japanese WolfGoatCabbage Puzzle Solve */
mtype = {Cop, Criminal, Mom, Dad, Girl, Boy, Boat};

#define DONE (r[Cop] == 1 && r[Criminal] == 1 && \
              r[Mom] == 1 && r[Dad] == 1 && \
              r[Girl] == 2 && r[Boy] == 2 && r[Boat] == 1)

#define NotWith(children, with) (r[children] == 2*(1-r[with]))
#define With(children, with)    (r[children] != 2*(1-r[with]))

#define CriminalUnsafe (r[Criminal] != r[Cop] && \
        (r[Criminal] == r[Mom] || r[Criminal] == r[Dad] || \
        With(Boy,Criminal) || With(Girl,Criminal) ))
#define BoysUnsafe ( With(Boy,Mom) && r[Mom]!=r[Dad] )
#define GirlsUnsafe ( With(Girl,Dad) && r[Mom]!=r[Dad] )


mtype prev_dr   = 0;
mtype prev_pass = 0;

inline printMove(driver, passenger, boat)
{
    if
        :: boat == 0 ->
            if
            :: passenger == 0 ->
                printf("%e goes there alone.\n", driver);
            :: else  ->
                printf("%e with %e go there.\n", driver, passenger);
            fi;
        :: else ->
            if
            :: passenger == 0 ->
                printf("%e goes back alone.\n", driver);
            :: else  ->
                printf("%e with %e go back.\n", driver, passenger);
            fi;
    fi;
```

```
}

inline update_r()
{
  r[driver] = r[driver] + (1-2*r[Boat]);
  if
     :: passenger != 0 -> r[passenger] = r[passenger] + (1-2*r[Boat]);
     :: else -> skip;
  fi;
  r[Boat] = 1 - r[Boat];
}

inline move(dr, pass)
{
  printMove(dr, pass, r[Boat]);
  if
    :: (dr == prev_dr && pass == prev_pass) -> printf("Don't do the same move!\n");
    :: else ->
        update_r();
        if
        :: (CriminalUnsafe || BoysUnsafe || GirlsUnsafe) ->
            /* undo move */
            update_r();
         :: else ->
            prev_dr = dr; prev_pass = pass;
        fi;
  fi;
}



/* Global array for positions, initially = 0 */
int r[8];
/* mtypes are assigned from 1, array are indexed from 0, so the r[0] is not used */

init {
    local mtype driver    = 0;
    local mtype passenger = 0;
```

```
    /* Run */
    do
      /* move Cop (with anyone or alone) */
      :: r[Cop] == r[Boat] -> driver = Cop;
        /* Choose a 'random' passenger if it is here */
        if
            :: r[Criminal] == r[Boat] -> passenger = Criminal
            :: r[Mom] == r[Boat] -> passenger = Mom
            :: r[Dad] == r[Boat] -> passenger = Dad
            :: With(Boy,Cop) -> passenger = Boy
            :: With(Girl,Cop) -> passenger = Girl
            :: true -> passenger = 0 /* no passenger at all */
        fi;
        move(driver, passenger);

      /* move Dad (with a Boy or with Mom or alone) */
      :: r[Dad] == r[Boat]  -> driver = Dad;
        if
            :: r[Mom] == r[Boat] -> passenger = Mom
            :: With(Boy,Dad) -> passenger = Boy
            :: true -> passenger = 0
        fi;
        move(driver, passenger);

      /* move Mom (with a Girl or alone) */
      :: r[Mom] == r[Boat]  -> driver = Mom;
        if
            :: With(Girl,Mom) -> passenger = Girl
            :: true -> passenger = 0
        fi;
        move(driver, passenger);

       :: DONE -> printf("SOLVED\n"); assert(0); break;
       :: else -> printf("WHAT?!\n"); assert(0); break; /* Should never happen! */
    od;
}
```

УДК: 004.02

Название: Spin для головоломок: Использование системы проверки моделей SPIN

для решения японской головоломки о переправе через реку и головоломки "Куб-1".

Автор(ы): Бодин Е.В. (Институт систем информатики СО РАН)

Аннотация: Статья описывает применение системы проверки моделей SPIN к решению японской головоломки о переправе через реку (продвинутый вариант задачи о волке, козе и капусте) и головоломки "Irregular IQ Cube" (также называемой "Куб-1").

Ключевые слова: SPIN, верификация