МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. М. В. Ломоносова

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

КАФЕДРА МАТЕМАТИЧЕСКОЙ ЛОГИКИ И ТЕОРИИ АЛГОРИТМОВ

Формальный язык спецификации процессов, ориентированный на использование аппарата уточнения для композиционного проектирования потоков работ

Дипломная работа
студента С. А. Ступникова

Руководители:
профессор, д. т. н. В. А. Сухомлин
профессор, д. ф.-м. н. Л. А. Калиниченко

МОСКВА, 2000 г.

Содержание

1 Введение

Разработка современных информационных и управляющих систем выполняется коллективами квалифицированных специалистов. Для обеспечения координации требований заказчика и действий разработчиков системы используются разнообразные модели и языки спецификаций, из которых семантически наиболее выразительными являются формальные языки.

Удобным механизмом формализации специфицирования является типизированная теория множеств, позволяющая, с одной стороны, моделировать достаточно сложные структуры данных и операции, манипулирующие этими структурами; а с другой стороны — быстро переходить от спецификации к программному коду. Именно на типизированной теории множеств построены такие языки, как Z, Object Z, AMN [14, 7, 10, 3].

Другой класс языков, ориентированных на средства управления процессами, представляют CSP, Timed CSP, CCS [2, 12, 13, 11]. Эти формализмы позволяют специфицировать параллельные системы, взаимодействующие в реальном времени. Важным примером области применения таких языков является развивающаяся технология потоков работ (workflows), основной движущей силой совершенствования которой являются растущие потребности процесса re-engineering, цель же бизнес-реинженерии — увеличение производительности, уменьшение стоимости и времени изменения окружения унаследованных систем, а также привлечение распределенных объектных технологий, технологий баз данных и т.д. в бизнес-процессы. Главная задача технологии потоков работ — отделение бизнес-процессов от существующих приложений. Дело в том, что код приложений является трудномодифицируемым, а бизнес-политика изменяется очень быстро. Другая проблема старых приложений — изолированность. Технология потоков работ призвана интегрировать распределенные приложения в рамках единого бизнес-процесса на высоком уровне абстракции [8].

Следующая отличительная особенность современных информационных систем заключается в тенденции перехода к их конструированию из готовых разнородных компонентов, распределнных в сети. Программист превращается в конструктора, собирающего, образно говоря, дом из кубиков. Большое значение при этом приобретает понятие уточнения спецификаций (refinement). Перед реализацией системы необходимо составить ее спецификацию, декомпозировать на части, а затем среди спецификаций ресурсов найти такие, которые бы уточняли части исходной спецификации.

Следовательно, нужны языки, формализующие понятие уточнения и технологии, позволяющие в некоторой степени автоматизировать процесс доказательства корректности уточнения. В качестве таких средств выступают Abstract Machine Notation (AMN) и B-technology [3, 4]. Именно поэтому AMN выбрана в качестве модели для СИНТЕЗ — язака проектирования и программирования систем из неоднородных ресурсов [1], призванного предоставить программисту максимум возможностей по проектированию систем, в том числе и из компонентов. В частности, в СИНТЕЗ существует каноническая модель определения потоков работ [9].

Проблема состоит в том, что в настоящее время не существует формального языка, сочетающего все необходимые программисту возможности: определение сложных структур данных в рамках типизированной теории множеств, формализацию уточнения и развитую технологию доказательства корректности уточнения, средства управления параллельными взаимодействующими процессами в реальном времени, когда к процессам приложима концепция компонентного проектирования: возможность конструирования искомого процесса из уже существующих; спецификации которых уточняют части исходной спецификации процесса, и факт уточнения является доказуемым. В языках процессного описания бедны средства представления структур данных, отсутствует реализованная технология доказательства корректности уточнения; а в AMN, совместно с B-технологией поддерживающей уточнение, отсутствуют средства управления процессами, необходимые, в частности, для отображения канонической модели потоков работ из СИНТЕЗ в AMN.

Продвижением в нужном направлении является csp2B tool, предоставляющий возможность специфицирования процессов с помощью подмножества операций CSP-нотации [5, 6]. csp2B конвертирует CSP-описание в стндартную B-машину (абстрактный тип данных AMN), при этом события из алфавита процесса интерпретируются как операции B-машины.

Подмножество операций конструирования процессов, поддерживаемое csp2B, крайне ограничено: в него включаются префиксирование, детерминированный выбор, условный оператор, процесс STOP , а также синхронная и асинхронная параллельная композиции на внешнем уровне.

В csp2B не рассматриваются последовательные процессы, позволяющие декомпозировать задачу на подзадачи, а в совокупности с параллельной композицией (которая должна быть определена не только на внешнем уровне) реализующие возможность распараллеливания и дальнейшего соединения в один поток некоторого процесса, (прием, необходимый для эффективных вычислений на мультипроцессорных платформах). Отсутствует поддержка сокрытия, без которого не обойтись, если необходимо абстрагироваться от некоторой функциональности системы. Такая абстракция является важным инструментом компонентного проекирования систем, когда часть исходной спецификации заменяется уточняющей ее спецификацией ресурса, в которой некоторые механизмы функционирования скрыты.

csp2B не поддерживает недетерминизм, одна из главных задач которого — абстрагироваться от конкретных деталей реализации. Недетерминированный процесс может быть реализован несколькими способами, каждый из которых демонстрирует те или иные особенности поведения. Таким образом достигается вы-

сокй уровень абстракции при описании поведения сложнах систем. Без поддержки недетерминизма также невозможно применение операции сокрытия.

В csp2B не рассматриваются прерывание, необходимое при описании исключительных ситуаций, специальная обработка которых является одной из парадигм современного программирования; а также временные операции, с помощью которых обеспечивается взаимодействие систем в реальном времени. Невозможно в csp2B и взаимодействие процессов между собой внутри одной спецификации: разрешается лишь взаимодействие с окружением.

Таким образом, в AMN, в совокупности с csp2B, наблюдается недостаточность средств управления процессами. Задачей работы ставится расширение AMN необходимыми возможностями: последовательными процессами, параллельностью на произвольном уровне, сокрытием, недетерминированным выбором и его вариантом — общим выбором (также возникающим при использовании сокрытия), прерыванием, организацией взаимодействия между процессами через каналы, основными временными примитивами Timed CSP — задержкой, таймаутом и временным префиксированием. Предполагается также предоставление возможности использования таких удобных сокращений CSP, как μ −оператор и множественный выбор, которые способствуют облегчению чтения спецификации и уменьшению ее текста. Расширение подразумевает представление механизмов AMN, которые будут реализовывать средства управления процессами, а также алгоритмов обработки процессного описания расширенной нотации. На основе этих алгоритмов будет работать тул, переводящий описание процессов в стандартную B-машину.

Данная работа проводилась в рамках проекта СИНТЕЗ, который посвящен исследованиям в области сред разнородных интероперабельных информационных ресурсов (HIRE). Одним из направлений проекта является разработка методов спецификации и повторного использования динамических аспектов HIRE — потоков работ.

Текст дипломной работы организован следующим образом. В разделе (2) рассматриваются возможности современных языков спецификаций. В разделе (3) разбираются возможности и принцип работы csp2B tool. В разделе (4) рассматриваются дополнительные операции, расширяющие AMN, а также примеры их использования. В разделе (5) приведенные в (4) операции интерпретируются средствами AMN и предлагаются алгоритмы обработки процессного описания с использованием этих операций. В разделе (7) рассматривается пример спецификации лифтовой системы и ее вариант, оттранслированный в AMN. В разделе (6) обсуждаются возможности применения аппарата уточнения AMN к расширенным спецификациям, что обеспечивает возможность перехода к композиционному проектированию процессов из процессных компонент. В разделе (8) приведены правила описания процессов в расширенной нотации. В заключении подведен итог выполненной работы.

2 Формальные языки и их возможности

Для специфицирования современных сложных программных систем необходимы формальные языки, совмещающие в себе следующие возможности:

1. Моделирование сложных структур данных и алгоритмов, которые включает в себя система.
2. Формализация понятия уточнения, наличие средств, позволяющих в некоторой степени автоматизировать процесс доказательства корректности уточнения, для получения в итоге не содержащего ошибок программного кода на некотором языке программирования.
3. Управление процессами, включая параллельность и взаимодействие в реальном времени, для моделирования переходов системы в различные состояния.

Проблема состоит в том, что существующие формальные языки специализированы на некоторых из вышеприведенных требований, и не позволяют реализовать их все целиком.

Object-Z — объектно-орентированное расширение Z. Это язык, основанный на типизированной теории множеств, позволяющий описывать структуры данных, в том числе и параметризованные, в виде классов; определять операции класса. Схемы операций могут композиционироваться в соответствии с законами схемного исчисления. В Object-Z нет понятия длительности операции и средств управления процессами, а потому использующие параллельность системы реального времени трудно реализуемы. Одним из расширений Object-Z, предназначенным для моделирования взаимодействия с окружением в реальном времени, является Timed Object-Z. Подход Timed Object-Z состоит из двух частей:

- Окружение моделируется как функции от времени, которые включаются в схему состояния класса.
- Существуют глобальные часы, моделирующие реальное время, включаемые в схему состояния класса как атрибут $now: \mathbb{T}$.

Значение атрибута *now* может увеличиваться в ходе операции, а следующая операция должна начинаться, как только закончилась предыдущая. Трудность состоит в том, что для исполнения операций в некотором желаемом порядке необходимо вычислять предусловия этих операций. Ослабление же предусловий для уточнения в Timed Object-Z недопустимо. Не очень удобно и взаимодействие с окружением через дополнительно введенные атрибуты класса.

Другие нотации, ориентированные именно на управление процессами – CSP и ее расширение Timed CSP, в котором наряду с обычными операциями над процессами – детерминированным и недетерминированным выбором, синхронной и асинхронной параллельной композицией, прерыванием etc., содержит следующие процессные и временные примитивы:

- Временное префиксирование
$$e \bullet t \rightarrow P(t)$$
Процесс может принять событие e, а время t, которое прошло с начала процесса до наступления события e, может стать параметром последующего поведения процесса P.
- Задержка
$$\textsc{Wait}\ t;$$
Процесс $\textsc{Wait}\ t;\ P$ не допускает никаких взаимодействий в течение времени t, после чего ведет себя как P.
- Таймаут
$$(e \rightarrow P) \rhd \{t\} Q$$
ведущий себя как P, но в случае непоявления события e в течение времени t, передающий управление Q.

Элегантность выражения последовательности событий, таймаутов и задержек в Timed CSP тем не менее омрачается трудной выразимостью сложных структур данных.

Формальным языком, объединяющим возможности Timed Object-Z и Timed CSP, является Timed Communicating Object Z. Подход TCOZ состоит в том, что схемы операций Z понимаются как конечные процессы (синтаксически и семантически); объекты, принадлежащие некоторым классам, также понимаются как процессы. Объектам разрешается иметь интерфейсы, реализуемые каналами из нотации CSP, посредством которых они (объекты) взаимодействуют друг с другом. Каналы являются глобальными и имеют тип *chan*, т.е. могут переносить сообщения любых типов.

В определение класса TCOZ могут входить определения процессов. Например, поведение объекта данного класса задается процессом *Main*, который определяется в схеме класса. Вообще, процессные термы конструируются посредством операций, включающих операции Timed CSP; а также с помощью схем операций, про которые известно, что они состоят из событий, изменяющих состояние объекта согласно предикату схемы. Точное определение этих событий не ставится задачей TCOZ. Длительность операции определяется значением переменной δ, которая неявно присутствует в каждой операционной схеме.

Классы могут содержать в качестве атрибутов другие классы, а также наследоваться.

Таким образом, широкие возможности TCOZ по моделированию сложных процессов позволяют считать TCOZ-спецификации своеобразным тестом, по которому мы можем судить о других нотациях: если удается перевести TCOZ-спецификацию на тестируемый формальный язык с приемлемыми затратами, то мы считаем его хорошим.

Недостаток нотации TCOZ состоит в том, что для нее понятие уточнения формализуется недостаточно хорошо: на данный момент не существует никаких средств, хотя бы в какой-то мере автоматизирующих доказательство корректности перехода от одного уточнения спецификации к другому. С другой стороны, существует нотация, также основанная на типизированной теории множеств – AMN – для которой есть соответствующие технологии (B-technology). Однако в AMN нет средств управления процессами. Попыткой совмещения CSP и B является csp2B tool.

3 csp2B tool

Тул переводит CSP-описания процессов, задающих поведение системы в стандартные B-машины, для которых могут быть сгенерированы proof obligations. При этом CSP-описание может являться уточнением другой CSP или B-машины. CSP-машина может также накладывать ограничения на порядок выполнения операций некоторой B-машины. Общий вид CSP-машины описан в [6].

Тул поддерживает следующие CSP-средства описания процессов:

- Префиксирование $\rightarrow P$, описывает процесс,вначале участвующий в событии e, а затем ведущий себя как P.
- Детерминированный выбор $(a \rightarrow P \,[]\, b \rightarrow Q)$, описывает процесс который вначале может участвовать либо в событии a, либо в событии b; а затем ведущий себя как P либо Q, в зависимости от того, какое событие наступило. В выборе может участвовать любое конечное множество событий:

$$(a_1 \rightarrow P_1 \,[]\, \ldots a_n \rightarrow P_n)$$

- Процесс STOP, описывает состояние deadlock.
- Условный оператор IF G THEN P ELSE Q, описывает процесс, ведущий себя как P либо Q, в зависимости от того, какое значение принимает B-предикат G.
- Параллельная композиция на внешнем уровне $P \parallel Q$

  PROCESS $P = P_1$
  CONSTRAINS $a_1 \ldots a_n$ WHERE
  ProcessDescriptions
  END
  PROCESS $Q = Q_1$
  CONSTRAINS $b_1 \ldots b_m$ WHERE
  ProcessDescriptions
  END

  Здесь *ProcessDescriptions* – набор рекурсивных определений, задающих процессы P_1 и Q_1, a_i, b_j – события, по которым происходит синхронизация процессов P и Q.

- Чередование нескольких индексированных экземпляров одного процесса на внешнем уровне

  PROCESS $P = |||u_1 \ldots u_n.P_1[u_1 \ldots u_n]$ WHERE
  ProcessDescriptions
  END

  Здесь $u_1 \ldots u_n$ – множество индексов чередующихся процессов.

События из алфавита могут быть параметризованы входными $In?x$ и выходными $Out!x$ параметрами (см. примеры из [5]), которые при переводе в B будут соответственно входными и выходными параметрами операций. В терминах CSP такие события будут входными и выходными каналами. Входные параметры также могут быть „точечными“. Это означает, что процесс может принять только указанное значение входного параметра (см. пример *Free* из [5]). Процессы также могут индексироваться, и эти индексы становятся переменными сгенерированной B-машины (см. пример *Idle* из [5]).

Теперь разберем, как же генерируется B-машина и какова семантика CSP в туле.

Процесс в нотации csp2B описывается набором формул вида

$$I_i(v) = P_i,$$

где I_i – идентификатор процесса, а P_i – процессный терм, который может содержать несколько идентификаторов процессов. Процессный терм P называется приведенным к *нормальной форме*, если $P =$ STOP или P имеет вид

$$P = Q_1 \,[]\, \ldots \,[]\, Q_n,$$

где каждый терм Q_i вида

$$\text{IF } G \text{ THEN } a \to I,$$

где I – идентификатор процесса. С помощью законов эквивалентных преобразований процессных термов из [5] можно привести все термы P_i из описания процесса к нормальной форме. Введем понятие пространства состояний машины – это будет декартово произведение множества процессных идентификаторов на тип индексирующих параметров, обозначим элементы пространства через $I(v)$. Для каждого события a из алфавита процесса может найтись несколько пар состояний $(I_i(v), I_i'(v')), i = 1 \dots k$; что в описании процесса есть фрагмент вида

$$I_i(v) = \dots [] \quad \text{IF } G(v) \text{ THEN } a \to I_i'(v') \quad [] \dots$$

Будем считать, что событие a переводит машину из состояния $I_i(v)$ в состояние $I_i'(v')$.

Покажем, как генерируется B-операция для произвольного события a из алфавита процесса. Пусть p – переменная, принимающая значения на множестве идентификаторов процессов, v – переменная, принимающая значения на множестве индексов процессов. Пусть для события a нашлось несколько описаний вида

$$I(v) = \dots [] \quad \text{IF } G(v) \text{ THEN } a.i?y!k \to I'(V(y)) \quad [] \dots, \tag{1}$$

где i, j, k – входные, выходные и точечные параметры события a соответственно. Тогда для события a будет создана B-операция вида

$$z \leftarrow a(x, y) \mathrel{\widehat{=}} S_a,$$

где S_a конструируется следующим образом: для каждого появления события a в нормализованном выражении вида (1), в B-операцию вставляется ветвь SELECT:

$$\text{SELECT } G(v) \wedge p = I \wedge x = i \text{ THEN } p, v, z := I', V(y), k \text{ END}$$

Различные ветви композиционируются оператором выбора из B $S [] T$. Параллельная композиция $P \parallel Q$ реализуется следующим образом: пусть – одно из событий, по которому синхронизируются P и Q; $S_{p1} \dots S_{pn}$ – ветви SELECT события a для процесса P; $S_{q1} \dots S_{qn}$ – ветви SELECT события a для процесса Q; тогда в результате обработки параллельной композиции S_a будет иметь вид

$$(S_{p1} [] \dots [] S_{pn}) \parallel (S_{q1} [] \dots [] S_{qn})$$

Недостаток csp2B состоит в том, что тул реализует далеко не все операции CSP над процессами, а параллельность и чередование разрешаются только на внешнем уровне. Невозможно также напрямую организовать взаимодействие между процессами через каналы. Таким образом, возникает задача: расширить тул дополнительными операциями и предложить механизмы реализации возможностей TCOZ.

4 Дополнительные операции

4.1 μ −оператор

Используя префиксирующий оператор, можно описывать полное поведение процесса, который рано или поздно останавливается. Однако, для определения процесса, потенциально работающего бесконечное время, необходим способ более сжатого описания повторяющихся действий. Желательно, чтобы такой способ не требовал знать заранее срок жизни объекта; это позволило бы описывать объекты,которые продолжали взаимодействовать с окружением до тех пор, пока в них есть необходимость. В CSP для этого используется рекурсивное определение процесса:

$$X = F(X) \tag{2}$$

Такой метод „самоназывания“будет правильно работать, только если в правой части уравнения рекурсивному вхождению имени процесса предшествует хотя бы одно событие [2]. В туле такое условие является необходимым при определении систем рекурсивных уравнений процессов. Удобно обозначать решение уравнения (2) через $\mu X \bullet F(X)$ (где X является связанной переменной); например, если мы хотим яснее представить себе смысл цикла:

$$\text{WHILE } G \text{ DO } Q = \mu X \bullet (\text{ IF } G \text{ THEN } (Q;\ X) \text{ ELSE SKIP })$$

Пример
Часы, которые могут бесконечно функционировать.

$$\text{PROCESS } P = \mathit{Clock} \text{ WHERE } \mathit{Clock} = \mu X \bullet (\mathit{tick} \to X) \quad \text{END}$$

4.2 Выбор по множеству событий

Когда при детерминированном выборе $(a \to P[]b \to Q)$ выполнено $P = F(a), Q = F(b)$ представляется удобным записать выбор следующим образом:

$$(x : B \to F(x)),$$

где $B = \{a, b\}$. Такого рода обзначения широко используются в CSP, способствуют сокращению текста и лучшему пониманию спецификации. Множество B в выборе по множеству может быть произвольным подмножеством алфавита.

Пример

Автомат, предлагающий чай или кофе за монету.

ALPHABET *Coffee* *Tea* *Coin*
ALPHABET_ SUBSET $B = \{Coffee, Tea\}$
PROCESS $P = Trade$
WHERE $Trade = \mu\, X \bullet (Coin \to (x : B \to X))$ END

4.3 Параллельная композиция с синхронизацией по множеству событий

Процесс определяется полным описанием его потенциального поведения. При этом часто имеется выбор между несколькими различными действиями. Этот выбор – какое из событий произойдет в действительности – может зависеть от окружения, в котором работает процесс. Однако, само окружение процесса может быть описано как процесс, поведение которого определяется в обычных терминах. Это позволяет исследовать поведение целой системы, состоящей из процесса и его окружения, взаимодействующих по мере их параллельного исполнения. Всю систему также следует рассматривать как процесс, поведение которого определяется в терминах поведения составляющих его процессов; эта система в свою очередь может быть помещена в еще более широкое окружение. Такой подход позволяет забыть разницу между процессами, окружениями и системами; все они – лишь процессы, поведение которых может быть описано единообразным способом.

Именно для описания сложных параллельных взаимодействующих процессов и предназначена операция параллельной композиции. Взаимодействия понимаются как события, требующие одновременного участия обоих процессов.

Проблема состоит в том, что параллельная композиция в csp2B разрешена только на внешнем уровне, т.е. не разрешается иметь выражения вида $e \to (P \parallel Q)$. Конечно, префиксирование дистрибутивно относительно параллельности

$$e \to (P \parallel Q) = (e \to P) \parallel (e \to Q),$$

но недетерминированный выбор(который мы введем позднее) в общем случае не дистрибутивен относительно параллельности:

$$P \sqcap (Q \parallel R) \neq (P \sqcap Q) \parallel (P \sqcap R)$$

и поэтому пронести через него параллельность на внешний уровень не удастся. Кроме того, любое применение закона дистрибутивности относительно параллельности ведет к снижению эффективности: разветвляется процесс, ранее исполнявшийся в одном потоке. При использовании параллельности только на внешнем уровне даже небольшие разветвления необходимо выносить в отдельный процесс, постоянно занимающий отдельный поток. Следовательно, нужно уметь интерпретировать паралельную композицию процессов с синхронизацией по некоторому множеству событий $P \mid \{e_1, \dots e_n\} \mid Q$ при произвольном вхождении в процессный терм. Подобный синтаксис удобен и не загромождает запись определения процесса в простых случаях. Когда же нам нужно построить систему параллельных процессов $Q_1, \dots Q_n$, каждая пара которых (Q_i, Q_j), где $i \neq j$, должна быть, возможно, синхронизирована друг с другом по некоторому множеству событий $\{e_{ij}^1 \dots e_{ij}^{k_{ij}}\}$; то запись подобной композиции через бинарную операцию параллельности неудобна. В этом случае предлагается использовать следующий синтаксис:

$P =\parallel (Q_1, \dots Q_n)$ WHERE
PROCESS $Q_1 = R_1$ CONSTRAINS $e_i \dots e_j$ WHERE
ProcessDescriptions
...
PROCESS $Q_n = R_n$ CONSTRAINS $e_k \dots e_l$ WHERE
ProcessDescriptions
END

Здесь для каждого процесса Q_i в секции CONSTRAINS указываются события, по которым он должен быть синхронизирован с другими процессами.

Пример

Рассмотрим пример автомата, торгующего напитками. Автомат загружается Pepsi, но если подвезут, может

торговать Cola. Известно, что за покупку любой продукции Cola награждает покупателя своей рекламной наклейкой.

MACHINE *TradeMachine*
ALPHABET *Coin* *Pepsi* *Cola* *Sticker*
PROCESS $S_{Trade} = Trade$ WHERE
$Trade = (Coin \to P) \sqcap$
$(Coin \to R \mid \{Cola, Pepsi\} \mid (Cola \to S \,[]\, Pepsi \to S))$
$P = Pepsi \to Coin \to P$
$R = (Cola \to Coin \to R \,[]\, Pepsi \to Coin \to R)$
$S = Sticker \to (Cola \to S \,[]\, Pepsi \to S)$
END
END

Ясно, что для этого примера

$$P \sqcap (R \parallel S) \neq (P \sqcap R) \parallel (P \sqcap S),$$

т.к. может получиться $P \sqcap S$ (Pepsi раздает рекламу). По этой же причине

$$P \sqcap (R \parallel S) \neq (P \sqcap R) \parallel S$$

Как видно, параллельность нельзя вынести на внешний уровень, пользуясь законами, не зависящими от вида процесса.

4.4 Секвенциальный оператор, SKIP

STOP определяется как процесс, не выполняющий никаких действий. Его нельзя назвать слишком полезным, и возникает он, скорее, в результате дедлока или других ошибок проектирования, нежели по замыслу разработчика. Существует, однако и другая причина, по которой процесс может прекратить работу, а именно — если он выполнил все, что ему полагалось. О таком процессе говорят, что он успешно завершился. Чтобы отличить такое завершение от STOP, удобно рассматривать его как специальное событие *Success*. *Последовательным* называется процесс, имеющий в алфавите *Success*. Процесс может только заканчиваться на событие *Success*, поэтому оно не может служить альтернативой множественного выбора:

$$(x : B \to P(x)) \quad \text{некорректно, если} \quad Success \in B$$

Определим SKIP как процесс, успешно завершающийся и больше ничего не делающий: $Prot(SKIP) = \{<>, < Success >\}$. При проектировании процесса для решения некоторой сложной задачи часто бывает полезно разбить ее на две подзадачи, одна из которых успешно завершается до начала другой. Если P – последовательный процесс, то последовательная композиция $P;\ Q$ есть процесс, ведущий себя сначала как P, а после его успешного завершения ведущий себя как Q. Если успешного завершения P не происходит, то не завершается и $(P;\ Q)$. При использовании секвенциального оператора возможно успешное решение задачи распараллелливания процесса, на некотором этапе требующего большого количества ресурсов и дальнейшего соединения всех параллельных подпроцессов в один поток.
Для корректного использования секвенциального оператора $(P;\ Q)$ все ветви (кроме циклических и оканчивающихся на STOP) должны оканчиваться на SKIP.
Пример
Рассмотрим пример верхнего этажа дома, в котором есть лифт. Нажав кнопку, человек запрашивает лифт по каналу *Enter* на свой этаж с номером *num*. Закрытие дверей лифта подтверждается событием *ServiseOk*, и затем кнопка гаснет.

MACHINE *TopFloor*
VARIABLES *num*, *Button*
INVARIANT $num \in Nat \wedge Button \in \{On, \quad Off\}$
ALPHABET *ServiceOk* *Request* $n \leftarrow Enter$
PROCESS $P = TopFloor$ WHERE
$TopFloor = \mu\, T \bullet (ServiceOk \to DownOff \,[]\, request \to PressDown);\ T$
$DownOff = Button := Off;$ SKIP
$PressDown =$ IF $Button = Off$ THEN
$Button := On;\ Enter!num \to$ SKIP ELSE SKIP
END
END

4.5 Присваивание

Одним из наиболее важных аспектов традиционного программирования является оператор присваивания. Естественно, что и в процессных языках спецификаций присваивание находит свое место: в том случае, если

изменение программной переменной входит в логику процесса, и его не требуется выносить как отдельную операцию, уместно использование оператора присваивания.
Итак, если x – программная переменная, e – выражение, а P – процесс, то $(x := e;\ P)$ есть процесс, ведущий себя как P, но только начальное значение x задается равным начальному значению e. Смысл присваивания задается следующим определением:

$$(x := e) \stackrel{def}{=} (x := e;\ \text{SKIP})$$

Для объявления переменных, с которыми будет оперировать csp-машина, введем в csp-машине раздел VARIABLES.
Пример использования оператора присваивания – машина *TopFloor*.

4.6 Операции, связанные с недетерминизмом

4.6.1 Недетерминированный выбор

Иногда процесс обладает некоторым спектром возможного поведения, но его окружение не имеет возможности ни влиять на выбор между различными альтернативами, ни даже наблюдать его. Такой недетерминизм в поведении процесса возникает вследствие сознательного решения не принимать во внимание факторы, определяющие выбор (особенно ярко это видно при использовании оператора сокрытия), а поэтому недетерминизм полезен для достижения высокого уровня абстракции при описании поведения систем.

Будем обозначать $P \sqcap Q$ процесс, который ведет себя или как P, или как Q, причем выбор между ними осуществляется произвольно, без контроля окружения, а значит, разработчик сможет реализовать этот процесс или как P, или как Q, принимая во внимание соображения, не имеющие отношения к спецификации (низкая стоимость, высокая скорость, etc.).
Пример применения недетерминированного выбора – машина *TradeMachine*.

4.6.2 Общий выбор

При недетерминированном выборе $P \sqcap Q$ окружение должно быть готово к тому, чтобы работать как с P, так и с Q, в то время как работа лишь с одним из этих процессов была бы, скорее всего, более простой. Более мудрой является общий выбор $P \between Q$ – операция, где окружение может управлять выбором между P и Q, при условии, что выбор будет сделан на первом же шаге. Заметим, что появление общего выбора неизбежно при использовании операции сокрытия.
Итак, если первое действие при общем выборе $P \sqcap Q$ не возможно для P, то выбирается Q; если Q не может выполнить первое действие, то выбирается P. Если же первое действие возможно как для P, так и для Q, то выбор между ними остается недетерминированным. Таким образом справедливы законы

$$(c \to P \between d \to Q) = (c \to P \,[]\, d \to Q) \quad \text{если } c \neq d \tag{3}$$

$$(c \to P \between c \to Q) = (c \to P) \sqcap (d \to Q) \tag{4}$$

Эти законы легко обобщаются на случай множественного выбора. Вопросом реализации тула является способ сведения выражения $P \between Q$, где P, Q – произвольные процессные термы, к множественному выбору и недетерминизму.
Пример
Рассмотрим два автомата по размену денег, которые может разменивать монеты достоинством 5, причем первый автомат разменивает монеты способами (3+2) и (3+1+1), а второй - способами (2+2+1) и (2+1+1+1); т.е.

$$A_1 = (c_5 \to c_3 \to c_2 \to A_1 \sqcap c_5 \to c_3 \to c_1 \to c_1 \to A_1)$$

$$A_2 = (c_5 \to c_2 \to c_2 \to c_1 \to A_2 \sqcap c_5 \to c_2 \to c_1 \to c_1 \to c_1 \to A_2)$$

Если мы хотим соединить функциональность автоматов, то можем взять композицию $A_1 \sqcap A_2$. Однако при этом мы вообще никак не сможем влиять на выбор и не будем уверены, что получим монету достоинством 3. Если же взять композицию $A_1 \between A_2$, то из дистрибутивности $\between$ относительно $\sqcap$ и (3), получим

$$\begin{aligned} A_1 \between A_2 \quad = \quad c_5 \to \quad & ((c_3 \to c_2 \to A_1 \,[]\, c_2 \to c_2 \to c_1 \to A_2) \sqcap \\ & (c_3 \to c_2 \to A_1 \,[]\, c_2 \to c_1 \to c_1 \to c_1 \to A_2) \sqcap \\ & (c_3 \to c_1 \to c_1 \to A_1 \,[]\, c_2 \to c_2 \to c_1 \to A_2) \sqcap \\ & (c_3 \to c_1 \to c_1 \to A_1 \,[]\, c_2 \to c_1 \to c_1 \to c_1 \to A_2)) \end{aligned}$$

Ясно, что в этом случае мы всегда сможем выбрать: иметь монету достоинством 3 или нет.

4.6.3 Сокрытие

Алфавит процесса, вообще говоря, содержит только те события, которые мы посчитали уместными и чье наступление требует одновременного участия обстановки. При описании внутреннего поведения механизма часто приходится иметь дело с событиями, отвечающими внутренним переходам этого механизма. Такие события могут отражать взаимодействия между параллельно работающими компонентами, из которых построен данный механизм. Создав механизм, мы делаем недоступной структуру его компонент, скрывая внутренние действия. Фактически мы хотим, чтобы эти действия происходили автоматически и в тот самый момент, когда для этого появляется возможность, и при этом были бы недоступны для контроля и наблюдения со стороны окружающего процесса.
Другая важная задача, решаемая с помощью операции сокрытия – ограничение функциональности компоненты при проектировании сложных систем из компонент. Имея спецификации системы, которую необходимо собрать из имеющихся ресурсов, а также спецификации этих ресурсов, необходимо декомпозировать исходную систему на части и найти среди спецификаций ресурсов такие, которые бы уточняли эти части. При этом возможна ситуация, когда часть функциональности компоненты, доступ к которой в исходной спецификации не определен, необходимо скрыть, т.е. удалить из описания процесса вызовы некоторых методов компоненты, подразумевая их внутренними.
Итак, если C – конечное множество событий, которое мы хотим скрыть, то $P \setminus C$ обозначает процесс, который ведет себя как P с той разницей, что все события из C скрыты.
Пример
Рассмотрим шумный торговый автомат, предлагающий напиток за монету. Автомат шумит при совершении операций и получает напитки со склада. Такие подробности покупателю знать совершенно незачем, к тому же автомат закрыт звукоизолирующим кожухом. При выдаче напитков на складе громко хлопает крышка.

PROCESS $S_{Trade} = (Trade \mid \{Cola\} \mid Store) \setminus \{Bang,\ Bung,\ GetCola\}$
WHERE
$Trade = Coin \rightarrow Bang \rightarrow GetCola \rightarrow Cola \rightarrow Bung \rightarrow Trade$
$Store = GetCola \rightarrow Lid \rightarrow Store$
END

4.7 Прерывание

При проектировании систем часто возникают ситуации, когда течение процесса необходимо прервать после наступления некоторого события, не принадлежащего алфавиту процесса. Вообще говоря, такое течение процесса является вариантом последовательной композиции, различие состоит в том, что передача управления процессу Q не зависит от успешного завершения процесса P. Обозначим эту ситуацию

$$P \triangledown (e \rightarrow Q),$$

это процесс, ведущий себя как P, пока не произошло событие e, далее управление передается процессу Q, а процесс P более не возобновляется. Использование оператора присваивания наиболее уместно при обработке всяческого рода исключительных ситуаций, учитывая повышенное внимание к исключениям в современных языках программирования. Пример использования оператора прерывания – процесс *Lifts* из машины *LiftSystem*.

4.8 Временные операции

Важной особенностью Timed CSP являются механизмы моделирования взаимодействия процесса с окружением в реальном времени. Такие возможности играют достаточно большую роль при проектировании сложных программных систем. Поэтому представляется разумным предложить вариант реализации временного префиксирования $e \bullet t \rightarrow P(t)$, задержки WAIT t; и таймаута $(e \rightarrow P) \rhd \{t\} Q$ в AMN, как основных временных примитивов, использующихся в Timed CSP.
Пример
В качестве примера, иллюстрирующего возможности CSP, связанные со временем, приведем *TimedCollection*.

MACHINE *TimedCollection*
SETS X
CONSTANTS $t_a, t_p, t_d, t_0, PurgeStale, FindOldest, GetX, GetTime$
PROPERTIES
$t_a \in \mathbb{T} \wedge t_p \in \mathbb{T} \wedge t_d \in \mathbb{T} \wedge t_0 \in \mathbb{T} \wedge$
$PurgeStale \in (\mathbb{T} \times \mathbb{F}(\mathbb{T} \times X) \rightarrow \mathbb{F}(\mathbb{T} \times X)) \wedge$
$PurgeStale(t, \varnothing) = \varnothing \wedge$
$\forall (t, s).(t \in \mathbb{T} \wedge s \in \mathbb{F}(\mathbb{T} \times X) \Rightarrow$
$PurgeStale(t, s) = \{t_0 \mapsto e \mid t_0 \in \mathbb{T} \wedge add(t_0, t) \mapsto e \in s\}) \wedge$
$FindOldest \in \mathbb{F}_1(\mathbb{T} \times X) \rightarrow (\mathbb{T} \times X) \wedge$

$\forall(s,t,e).(s \in \mathbb{F}_1(\mathbb{T} \times X) \wedge t \in \mathbb{T} \wedge e \in X \Rightarrow$
$(t = \min(\mathrm{dom}(s)) \wedge t \mapsto e \in s \Rightarrow \mathit{FindOldest}(s) = t \mapsto e)) \wedge$
$\mathit{GetX} \in \mathbb{T} \times X \to X \wedge$
$\forall(s,t,e).(s \in \mathbb{T} \times X \wedge t \in \mathbb{T} \wedge e \in X \Rightarrow$
$(t \mapsto e = s \Rightarrow \mathit{GetX}(s) = e))$
$\mathit{GetTime} \in \mathbb{T} \times X \to \mathbb{T} \wedge$
$\forall(s,t,e).(s \in \mathbb{T} \times X \wedge t \in \mathbb{T} \wedge e \in X \Rightarrow$
$(t \mapsto e = s \Rightarrow \mathit{GetTime}(s) = t))$
VARIABLES s, p_t
INVARIANT $s \in \mathbb{F}(\mathbb{T} \times X) \wedge p_t \in \mathbb{T} \times X$
INITIALISATION
$s := \varnothing \parallel$
ANY p WHERE $p \in \mathbb{T} \times X$ THEN $p_t := p$ END
ALPHABET $\mathit{Left}(e : X) \quad e \leftarrow \mathit{Right}$
PROCESS $S_P = TC$ WHERE
$TC =$
IF $s = \varnothing$ THEN
$\mathit{Left}?e \rightarrow$ WAIT t_a; $s := s \cup \{t_0 \mapsto e\}$; TC
ELSE
$p_t := \mathit{FindOldest}(s)$;
$((\mathit{Left}?e \bullet t_l \rightarrow$ WAIT t_a; $s := \mathit{PurgeStale}(\mathit{add}(t_l, t_a), s) \cup \{t_0 \mapsto e\}$; $TC) \triangleright$
$\{\mathit{GetTime}(p_t)\}$ WAIT t_p; $s := \mathit{PurgeStale}(\mathit{add}(\mathit{GetTime}(p_t), t_p), s)$; $TC)$ []
$(\mathit{Right}!\mathit{GetX}(p_t) \bullet t_r \rightarrow$ WAIT t_d; $s := \mathit{PurgeStale}(\mathit{add}(t_r, t_d), s)) \triangleright$
$\{\mathit{GetTime}(p_t)\}$ WAIT t_p; $s := \mathit{PurgeStale}(\mathit{add}(\mathit{GetTime}(p_t), t_p), s)$; $TC))$
END
END .

TimedCollection представляет собой хранилище элементов множества X, каждый из которых помечен временем, которое ему осталось жить в коллекции. Максимальное время жизни элемента в коллекции t_0: именно такое время ставится в соответствие элементу, прибывающему по каналу *Left*. Периодически применяющаяся операция *PurgeStale* удаляет из коллекции устаревшие элементы и уменьшает время жизни оставшихся. Старейший элемент может быть забран из коллекции по каналу *Right*. Операции добавления элемента в коллекцию, удаления и чистки имеют время работы, определяемое константами t_a, t_d, t_p соответственно. В случае, если коллекция пуста, процесс может лишь получить элемент по каналу *Left*, иначе возможно взаимодействие по обоим каналам. В случае отсутствия взаимодействия в течение времени жизни старейшего из элементов, определяемого функцией *FindOldest*, происходит обновление коллекции.

4.9 Взаимодействие между процессами

Взаимодействие между процессами состоит в передаче сообщений по каналам, при этом каждая передача сообщения является событием. Механизм взаимодействия процессов с окружением реализован в csp2B с помощью параметризованных событий, однако если мы попытаемся организовать взаимодействие между параллельными процессами через канал, то передачи сообщения не будет.

ALPHABET $m \leftarrow \mathit{Channel}(x : \mathit{MESSAGE})$
PROCESS $S_{\mathit{From}} = \mathit{From}$
CONSTRAINS $m \leftarrow \mathit{Channel}$ WHERE
$\mathit{From} = \mathit{Channel}!msg \rightarrow \mathit{From}$
END
PROCESS $S_{\mathit{To}} = \mathit{To}$
CONSTRAINS $\mathit{Channel}$ WHERE
$\mathit{To} = \mathit{Channel}?x \rightarrow \mathit{Receive}(x)$
$\mathit{Receive}(y : \mathit{MESSAGE}) = \mathit{Channel}?x \rightarrow \mathit{Receive}(x)$
END

В данном примере параллельные процессы *From* и *In* синхронизованы по каналу *Channel*: *From* посылает, а *In* принимает сообщения. В действительности операция *Channel* будет выглядеть следующим образом

$m \leftarrow \mathit{Channel}(x) \mathrel{\widehat{=}}$
pre $x \in \mathit{MESSAGE}$ THEN
SELECT $S_{\mathit{From}} = \mathit{From}$ THEN $m, S_{\mathit{From}} := msg, \mathit{From}$ END
$\parallel$
(SELECT $S_{\mathit{To}} = \mathit{To}$ THEN $x_1, S_{\mathit{To}} := x, \mathit{Receive}$ END []

SELECT $S_{To} = Receive$ THEN $x_1, S_{To} := x, Receive$ END)
END

Видно, что сообщения, передаваемые *From* никак не связаны с сообщениями, которые передает *To*, несмотря на синхронизацию этих процессов. Таким образом, имеющимися средствами csp2B нельзя напрямую организовать взаимодействие между процессами. Поэтому введем понятие канала: разрешим в csp-машине секцию CHANNELS , в которой будем указывать имена каналов и тип передаваемого сообщения:

CHANNELS $C_1(t_1 : T_1) \dots C_n(t_n : T_n)$

Процессы могут синхронизироваться по каналам: если в вышеприведенном примере имя *Channel* указать в секции CHANNELS , то процессы действительно будут синхронизированы по этому каналу в том смысле, что *To* будет получать посланные *From* сообщения. По соглашению, принятому в CSP, каналы передают сообщения только в одну сторону. Подробности интерпретации каналов будут приведены в разделе (5).

5 Обработка дополнительных операций

В этом разделе работы мы рассмотрим вопросы реализации тула, который должен переводить csp-машину в стандартную B-машину. Определение процессов в csp-машине при этом может содержать дополнительные операции, введенные в разделе (4).

Основными задачами тула являются:

- определение пространства состояний системы S,
- определение отношения перехода $transition(P)$ на тройках (e, I, J), где – событие, I и J – различные состояния машины, – процесс,
- определение ветвей SELECT , которые должны входить в операции, соответствующие различным событиям.

Смысл отношения перехода следующий:

$$transition(P) = \text{true} \Leftrightarrow I \xrightarrow{e} J \in P,$$

т.е. процесс P содержит вхождение события e, которое переводит систему из состояния I в состояние J. Тела операций, соответствующих событиям, должны формироваться согласно этому отношению: операция события e обязана содержать ветвь

$$\text{SELECT } S = I \text{ THEN } S := J \text{ END}$$

В разделе (3) описано, каким образом csp2B решает данные задачи, приводя описание процессов к нормальной форме и используя следующие законы эквивалентного преобразования процессных термов:

IF G THEN P ELSE Q = (IF G THEN P) [] (IF $\neg$ G THEN Q)
IF G THEN (P [] Q) = (IF G THEN P) [] (IF G THEN Q)
IF G THEN (IF H THEN P) = (IF $G \wedge H$ THEN P)
IF G THEN STOP = STOP
STOP [] $P = P$

где P, Q – процессные термы, G, H – выражения AMN. Префиксирование нормализуется в csp2B заменой выражения

$$I(v) = (\text{ IF } G \text{ THEN } a \to P(v, w)) \,[]\, Q$$

где P – процессный терм, а w – входные параметры события a; на пару выражений

$$I = (\text{ IF } G \text{ THEN } a \to J(v, w)) \,[]\, Q$$

$$J(v, w) = P(v, w)$$

Естественно, что тул, обрабатывающий наряду с обычными операциями csp2B введенные в разделе (4) операции, должен использовать эти преобразования при первой же возможности. Появление группы дополнительных операций требует описания интерпретации этих операций.

Интерпретация же может осуществляться сведением к средствам csp2B, механизмам AMN либо операциям, которые интерпретированы первыми двумя способами. При этом нужно показывать, каким образом решаются поставленные выше задачи тула. Рассмотрим теперь поочередно введенные в разделе (4) операции и их интерпретацию.

5.1 μ −оператор

Рассмотрим некоторый процесс Q:

PROCESS $S_Q = Q$ WHERE
ProcessDescription

μ −оператор может использоваться при определении процесса Q двумя способами:

- выражение вида $\mu\, P \bullet F(P)$ встречается внутри процессного терма, входящего в *ProcessDescription*
- в *ProcessDescription* входит строка вида

$$I = \mu\, P \bullet F(P) \tag{5}$$

В первом случае заменим выражение $\mu\, P \bullet F(P)$ на идентификатор P_μ, который будет являться состоянием машины и добавим в *ProcessDescription* строку

$$P_\mu = F(P_\mu)$$

При этом следует наложить естественное ограничение, состоящее в том, что рекурсивному вызову процесса P через μ −оператор должно предшествовать хотя бы одно событие (такое же ограничение всегда накладывает csp2B). Во втором случае будем интерпретировать строку (5) как

$$I = F(I)$$

5.2 Выбор по множеству событий

Рассмотрим процессный терм вида $(x : B \to P(x))$, где $B = \{a_1, \dots a_n\}$ – подмножество алфавита процесса, - процессный терм, зависящий от x. Заведем в B-машине в секции SETS множество $Alphabet = \{a_1, \dots a_m\}$ всех событий алфавита, разрешим секцию ALPHABET_ SUBSET в csp-машине, где будем определять подмножества алфавита:

ALPHABET_ SUBSET $B = \{a_1, \dots a_n\}$

Вышеприведенный терм будем интерпретировать как

$$(a_1 \to P(a_1) \,[]\, \dots \,[]\, a_n \to P(a_n))$$

Таким образом, множественный выбор и μ −оператор сведены к средствам csp2B. Рассмотрим теперь операцию, которая сводится к параллельной композиции — прерывание.

5.3 Прерывание

Будем интерпретировать $P \triangledown (e \to Q)$ как

$$P \parallel (e \to Q)$$

с условием, что после того, как произойдет событие e, процесс P будет остановлен. Это означает, что в операции, соответствующей событию e, будет следующая ветвь SELECT:

$$\text{SELECT } S_Q = I_{e \to Q} \text{ THEN } S_Q, S_P := Q, \mathit{NotActivated} \text{ END},$$

где S_P, S_Q — переменные состояния параллельных ветвей нашего процесса, $I_{e \to Q}$ — идентификатор процесса $(e \to Q)$. Такая пара переменных и идентификатор будет заводиться всякий раз при рапараллеливании процесса, и инициализация этих переменных будет происходить в событии, префиксирующем параллельную композицию. Подробнее данный механизм будет рассмотрен при описании интерпретации параллельной композиции.
Если процесс P является последовательным, то в ветви события *Success*, отвечающие за переходы P в SKIP следует добавлять подстановку

$$I_{e \to Q} := \mathit{NotActivated}$$

5.4 Временные операции

Для интерпретации временных операций необходимы специальные средства AMN, чтобы в дальнейшем свести эти операции к параллельной композиции и средствам csp2B.

Заведем B-машину *Time*, в которой определим тип времени, переменную текущего времени *now*, отношение сравнения временных переменных *before*, функцию, выдающую текущее время *CurrentTime*, функцию сложения временных переменных *add* и операцию обновления текущего времени *Update_Act*. При реализации спецификации, использующей временные операции, разработчик будет уточнять все вышеприведенные

структуры в зависимости от архитектуры или языка программирования.

MACHINE *Time*
SETS $\mathbb{T}$
CONSTANTS t_0, *before*, *add*, *CurrentTime*
PROPERTIES
$t_0 \in \mathbb{T} \wedge$
$before \in \mathbb{T} \leftrightarrow \mathbb{T} \wedge$
$\forall(t_1, t_2, t_3).(t_1 \in \mathbb{T} \wedge t_2 \in \mathbb{T} \wedge t_3 \in \mathbb{T} \Rightarrow$
$(t_1 \mapsto t_2 \in before \Rightarrow t_2 \mapsto t_1 \notin before) \wedge$
$(t_1 \mapsto t_2 \notin before \Rightarrow t_2 \mapsto t_1 \in before) \wedge$
$t_1 \mapsto t_1 \notin before \wedge$
$(t_1 \mapsto t_2 \in before \wedge t_2 \mapsto t_3 \in before \Rightarrow t_1 \mapsto t_3 \in before)) \wedge$
$CurrentTime \in \mathbb{T} \rightarrow \mathbb{T} \wedge$
$\forall t.(t \in \mathbb{T} \Rightarrow t \mapsto CurrentTime(t) \in before) \wedge$
$add \in \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{T} \wedge$
$\forall(t_1, t_2, t_3).(t_1 \in \mathbb{T} \wedge t_2 \in \mathbb{T} \wedge t_3 \in \mathbb{T} \Rightarrow$
$add(t_1, t_2) = add(t_2, t_1) \wedge$
$add(add(t_1, t_2), t_3) = add(add(t_1, t_3), t_2) \wedge$
$t_1 \mapsto add(t_1, t_2) \in before)$
VARIABLES *now*
INVARIANT $now \in \mathbb{T}$
INITIALISATION $now := t_0$
OPERATIONS
$Update\_Act \mathrel{\widehat{=}} now := CurrentTime(now)$
END

Как видно из свойств, отношение *before* антисимметрично, нерефлексивно и транзитивно.

В дальнейшем мы будем использовать операцию *Update_Act* как событие для реализации временных операций, поэтому в csp-машину неявно будем включать *Time* с помощью CONJOINS . Далее в csp-машине, использующей временные операции, также неявно определим следующий процесс

PROCESS $S_{Time} = Time$ CONSTRAINS *Update* WHERE
$Time = \mu\, X \bullet Update \rightarrow X$
END

Очевидно, он будет параллелен основному процессу csp-машины, назовем его P.

PROCESS $S_P = P$ CONSTRAINS *Update* WHERE
ProcessDescriptions
END

Для синхронизации *Time* с основным процессом используется указание в CONSTRAINS -части события *Update*; если же основной процесс также распараллелен, то CONSTRAINS *Update* ставится в каждом из определений процесса. При этом процессы синхронизируются по *Update* только с процессом *Time*, но не между собой.
Если в спецификации существует несколько композиций чередующихся процессов, параметризованных, соответственно, индексами $v_1 \ldots v_m$, использующих в своих *ProcessDescription* временные операции, то событие *Update* также нужно параметризовать этими индексами $Update[v_1 \ldots v_m]$, а в предусловии операции *Update* необходимо предусмотреть возможность принятия индексами специального значения *NULL*. Если v_i при вызове *Update* равен *NULL*, это означает, что процесс-композиция, параметризованный v_i не принимает участия в данном переходе.

$Update(v_1 \ldots v_m) \mathrel{\widehat{=}}$
PRE $v_1 \in V_1 \cup \{NULL\} \wedge \ldots \wedge v_m \in V_m \cup \{NULL\}$
SELECT $v_1 \neq NULL \wedge SP(v_1) = I_1$ THEN $SP(v_1) = J_1$ END
...
SELECT $v_m \neq NULL \wedge SQ(v_m) = I_m$ THEN $SQ(v_m) = J_m$ END
...
END

Таким образом, каждая из ветвей SELECT будет менять состояние только одного процесса из одной композиции чередования.

Разберем теперь, каким образом реализуются временные операции.

5.4.1 Временное префиксирование

Рассмотрим некоторый процесс Q

PROCESS $S_Q = Q$ WHERE
ProcessDescription

в *ProcessDescription* которого входит терм

$$e \bullet t \to P(t) \tag{6}$$

$P(t)$ здесь также процессный терм, а временная переменная t выступает в роли индекса процесса, поэтому нужно завести переменную $T_1 \in \mathbb{T}$, в которой будет хранится время, прошедшее с начала процесса (6) до наступления события e (эта переменная добавится к множеству индексов процесса Q). Время начала процесса необходимо зафиксировать, т.е. предварить терм (6) присваиванием $T_0 := now$, где $T_0 \in \mathbb{T}$. Выражение (6) следует заменить на

$$e \bullet t \to I_P(t)$$

где I_P – процессный идентификатор, который нужно добавить к множеству состояний процесса Q; а также добавить в *ProcessDescription* строку

$$I_P = P(t)$$

В операцию, соответствующую событию e, нужно добавить следующую ветвь SELECT :

$$\text{SELECT } S = I \text{ THEN } T_1 := now - T_0 \parallel S := I_P \text{ END}$$

Здесь S – переменная состояния процесса, а I – состояние процесса Q перед сдачей управления процессу (6). Состояние I можно считать определенным, т.к. процессный терм разбирается начиная с внешних операций, и если обработчик перешел к терму (6), это означает, что состояние, из которого процесс входит в (6) уже фиксировано.

5.4.2 Задержка

Рассмотрим процесс

$$Q = \text{Wait } t;\ P$$

В течение времени t, процесс Q не должен реагировать ни на какие сообщения, а затем передать управление процессу P. Зафиксируем время начала Q в переменной T_w предварением процесса Q присваиванием $T_w := now$, а выражение

$$\text{Wait } t;$$

будем интерпретировать как процесс, регулярно проверяющий, не закончилось ли время ожидания. Если время ожидания вышло, то управление передается процессу P через SKIP :

$$\text{Wait } t;\ P \rightsquigarrow T_w := now;\ \mathit{Update} \to R;\ P,$$

где

$$\begin{array}{rcll} R & = & \mu\, X \bullet & \text{IF } now \mapsto add(T_w, t) \in \mathit{before} \\ & & & \text{THEN } \mathit{Update} \to X \\ & & & \text{ELSE } \mathit{Update} \to \text{SKIP} \end{array}$$

Если Q индексирован некоторыми параметрами $Q(v)[u]$, то R также будет индекстрован этими параметрами $R(v)[u]$.

5.4.3 Таймаут

Рассмотрим процесс

$$R = (e \to P) \rhd \{t\} Q$$

Изначально он должен вести себя как $(e \to P)$, но в случае непоявления события e в течение времени t обязан передать управление Q. Будем хранить время начала R в переменной $T_t \in \mathbb{T}$, а процесс R интерпретировать как

$$T_t := now;\ \mathit{Update} \to ((e \to \text{SKIP}) \parallel S);\ P,$$

где

$$S = \mu\, X.(\text{ IF } G \text{ THEN } Q \text{ ELSE } \mathit{Update} \to \text{SKIP });\ X,$$

а

$$G = now \mapsto add(T_t, t) \in \mathit{before}$$

Если R индексирован некоторыми параметрами $R(v)[u]$, то S также будет индекстрован этими параметрами $S(v)[u]$. Ситуацию

$$((e \to P) \rhd \{t\} Q);\ U,$$

когда после своего успешного завершения процесс Q должен передать управление процессу U; нейтрализуем законом

$$((e \to P) \rhd \{t\} Q);\ U = (e \to P;\ U) \rhd \{t\}(Q;\ U)$$

Процесс обработки таймаута легко обобщается на случай множественного выбора:

$$(e : B \to P(e)) \rhd \{t\} Q \rightsquigarrow T_t := now;\ ((e : B \to \text{ SKIP }) \parallel S);\ P(e),$$

где B – некоторое множество событий, а S и T_t имеют тот же смысл, что и при однократном выборе.

Рассмотрим теперь операции, сводящиеся к недетерминизму — сокрытие и общий выбор. При их анализе будем иметь ввиду, что все операции, приведенные выше в разделе (5), мы уже проинтерпретировали, и можно считать, что в тексте их нет.

5.5 Сокрытие

Сокрытие — такая операция, при которой скрываемые события становятся неэаметны для окружения, т.е. исчезают из описания процесса. При этом может возникнуть непредваренная рекурсия:

$$P = (c \to P) \setminus \{c\} \tag{7}$$

Решением уравнения (7) является расходящийся процесс CHAOS [2]. Расходящиеся процессы ведут себя непредсказуемо и потому не поддаются спецификации. Отсюда следует, что при использовании сокрытия тулу придется проверять, не нарушена ли предваренность рекурсии. Алгоритм, проверяющий предваренность рекурсии, должен работать параллельно с обработкой csp-текста, поэтому мы разберем его после интерпретации операций.

Рассмотрим теперь алгоритм обработки процессного терма $P \setminus C$, где P – процессный терм, а C – множество событий $\{e_1 \dots e_n\}$, подлежащих сокрытию (множество C можно определить в секции ALPHABET_ SUBSET , однако допустима запись $P \setminus \{e_1 \dots e_n\}$). Этот алгоритм также должен работать параллельно с обработкой csp-текста.

РазборСокрытия$(P,)$
если $C = \varnothing$ **то конец** согласно закону $P \setminus \varnothing = P$
если $\neq \varnothing$ **то**
если $P =$ STOP **или** $P =$ SKIP **то конец**
согласно законам
STOP $\setminus C =$ STOP
SKIP $\setminus C =$ SKIP
если $P = I$, где I – идентификатор процесса P_I **то**
помечаем, что в процессном терме P_I нужно сокрыть события из C.
Такая пометка аналогична замене $I = P_I$ на $I = (P_I) \setminus C$
конец
если $P = Q \setminus B$ **то** **РазборСокрытия**$(Q, \cup B)$
согласно закону $(P \setminus B) \setminus C = P \setminus (B \cup C)$
конец
если $P = (R \sqcap S)$ **или** $P = (R;\ S)$ **или** $P = (R \,\text{Ⅱ}\, S)$ **или** $P = (x := e;\ R)$
или $P =$ IF G THEN R ELSE S **то**
РазборСокрытия$(R,)$
РазборСокрытия$(S,)$
конец
если $P = (b_1 \to R_1 \,[]\, \dots \,[]\, b_m \to R_m)$ **и** $\{b_1 \dots b_m\} \cap C = \varnothing$ **то**
РазборСокрытия$(R_1,)$
$\dots$
РазборСокрытия$(R_m,)$
согласно закону $B \cap C = \varnothing \Rightarrow (x : B \to P(x)) \setminus C = (x : B \to (P(x) \setminus C))$
конец
если $P = (b_1 \to R_1 \,[]\, \dots \,[]\, b_m \to R_m)$ **и** $\{b_1 \dots b_m\} \cap C \neq \varnothing$ **то**
выражение $(P \setminus C)$ заменяется согласно закону
$(x : B \to P(x)) \setminus C = Q \sqcap (Q \,\text{Ⅱ}\, (x : (B \setminus C) \to P(x)))$,
где $Q = \sqcap_{x:B \cap C}(P(x) \setminus C)$ [2] и запускаются
Сокрытие(R_i, b_i)
$\dots$
Сокрытие(R_j, b_j),
РазборСокрытия$(R_i,)$
$\dots$
РазборСокрытия$(R_j,)$,

где $\{b_i \dots b_j\} = \{b_1 \dots b_m\} \cap C$
конец
если $P = R \mid \{a_1 \dots a_m\} \mid S$ **и** $\{a_1 \dots a_n\} \cap C = \varnothing$ **то**
РазборСокрытия$(R,)$
РазборСокрытия$(S,)$
конец
если $P = R \mid \{a_1 \dots a_m\} \mid S$ **и** $\{a_1 \dots a_n\} \cap C \neq \varnothing$ **то**
РазборСокрытия$(R,)$
РазборСокрытия$(S,)$
СокрытиеСинхронизации$(R \mid \{a_1 \dots a_m\} \mid S, C)$
конец
конец

Алгоритм **Сокрытие** подклеивает скрываемое событие b к терму R.
Пусть скрываемое вхождение b переводит процесс из состояния I в состояние J. Заметим, что событие b может выполнять различную дополнительную работу, кроме перевода S_M из состояния в состояние. Это может быть присваивание внутренних переменных или вызов операции из CONJOINS -машины. Обозначим подстановку, совершающую всю эту дополнительную работу в событии b через L, а подстановку, изменяющую внутренние переменные – через W.

Сокрытие(R, b)
если $R =$ STOP **то конец**
если $R =$ SKIP **то**
добавим в операцию *Success* следующую ветвь SELECT :

$$\text{SELECT } S_M = I \text{ THEN } S_M := N \parallel L \text{ END},$$

где N – состояние, в которое процесс входит из J через SKIP .
конец
если $R = I'$, где I' – идентификатор процесса $P_{I'}$ **то**
множественного выбора $(b \to I' \,[]\, \dots)$ быть не может, т.к. иначе после сокрытия будет нарушена предваренность рекурсии, а следовательно имеется событие c, префиксирующее b:

$$c \to b \to I'$$

Добавим к телу c в качестве параллельной подстановки L.
конец
если $R = Q \setminus B$ **то** **Сокрытие**(Q, b) **конец**
если $R = (P \sqcap S)$ **или** $R = (P \,[\!]\, S)$ **или** $R = (P \parallel S)$ **то**
Сокрытие(P, b)
Сокрытие(S, b)
конец
если $R = (P;\ S)$ **то**
Сокрытие(P, b)
конец
если $R =$ IF G THEN P ELSE S **то**
Сокрытие(P, b)
Сокрытие(S, b),
а G следует заменить на $[W]G$, поскольку в G могут встречаться переменные, изменяемые в L.
конец
если $R = (x := e;\ S)$ **то**
Сокрытие(S, b),
а $x := e$ следует заменить на $x := [W]e$, поскольку в e могут встречаться переменные, изменяемые в W.
конец
если $R = (b_1 \to R_1 \,[]\, \dots \,[]\, b_m \to R_m)$ **то**
добавим в тело каждого b_i ветвь SELECT

$$\text{SELECT } S_M = I \text{ THEN } S_M := I_{R_i} \parallel L \parallel L_i \text{ END},$$

где I_{R_i} – идентификатор R_i, L_i – подстановка из b_i соответствующая переходу $J \xrightarrow{b_i} I_{R_i}$, в которой все выражения e_i, стоящие в правой части оператора присваивания, заменены на $[W]e_i$.
конец

конец

СокрытиеСинхронизации$(R \mid \{a_1 \dots a_m\} \mid S, C)$
Рассмотрим для простоты случай $(R \mid \{a\} \mid S) \setminus \{a\}$: при большем количестве событий рассуждения для каждого из них будут аналогичны рассуждениям для события a; обработку всех таких событий, подлежащих сокрытию и синхронизации, будем проводить последовательно.

Будем обрабатывать операции, составляющие процессные термы P и Q обычным порядком, помня при этом, что нам нужно синхронизировать P и Q по событию a, и затем скрыть его. В процессе обработки найдем все пары состояний процесса P, $(S_P^{1i}, S_P^{2i}), i = 1 \dots n$; что a переводит P из состояния S_P^{1i} в состояние S_P^{2i}, а также аналогичные пары $(S_Q^{1j}, S_Q^{2j}), j = 1 \dots m$ для процесса Q. Далее для каждого вхождения события a в термы P и Q найдем терм R, который префиксируется событием a, а затем проведем **Сокрытие**(R, a). Однако для того, чтобы синхронизировать P и Q по a, необходимо учитывать параллельные переходы при подклеивании функциональности a к другим событиям в процессе работы алгоритма **Сокрытие**. Пусть, например, переход $I_1 \xrightarrow{a} I_2$ подклеивается к переходу $I_2 \xrightarrow{b} I_3$ процесса P; функциональность a задается подстановкой F_a, функциональность b – подстановкой F_b, внутренние переменные изменяются в a подстановкой W_a. В таком случае в тело b нужно вставить ветвь SELECT

$$
\begin{array}{l}
\quad \text{SELECT } S_P = I_1 \text{ THEN } S_P := I_3 \parallel F_a \parallel F_b' \text{ END} \\
\quad \parallel \\
(\ \text{SELECT } S_Q = S_Q^{11} \text{ THEN } S_Q := S_Q^{21} \text{ END } [] \\
\quad \dots \\
\quad \text{SELECT } S_Q = S_Q^{1m} \text{ THEN } S_Q := S_Q^{2m} \text{ END }\ )
\end{array}
$$

где F_b' есть F_b, в которой все выражения e_j, стоящие в правой части оператора присваивания, заменены на $[W_a]e_j$.
Заметим, что в теле любого события , которое переводит Q из состояния S_Q^{2k} в состояние S_Q^{3k} должны быть как ветви, содержащие в себе внутренний переход $S_Q^{1k} \xrightarrow{a} S_Q^{2k}$, так и ветви, содержащие ровно переход $S_Q^{2k} \xrightarrow{a} S_Q^{3k}$. Это необходимо для того, чтобы после явного события из P, содержащего в себе функциональность a, Q совершил лишь переход, предписанный a, а затем мог бы действовать по своему усмотрению.
Аналогичные рассуждения следует провести для всех вхождений a в P и Q.
конец

С помощью этих алгоритмов сокрытие исчезает из csp-текста, таким образом его(сокрытие) в дальнейших рассуждениях можно не принимать во внимание.

5.6 Общий выбор

В разделе (4.6.2) были показаны два закона, сводящие общий выбор к обычному выбору и недетерминированному выбору. Эти законы обобщаются на случай произвольного множественного выбора следующим образом:

$$
\begin{array}{l}
(x : A \to P(x)) \,[\!]\, (y : B \to Q(y)) = \\
\quad (z : (A - B) \to P(z) \,[]\, z : (B - A) \to Q(z) \,[]\, z : (A \cap B) \to (P(z) \sqcap Q(z))
\end{array}
$$

Рассмотрим, каким образом с помощью этого **обобщенного** закона можно обрабатывать любые термы $(P \,[\!]\, Q)$. Общая идея состоит в том, чтобы, пронося некоторым образом различные операции через общий выбор, свести термы P и Q к множественному выбору (или к префиксированию, которое является частным случаем множественного выбора). Разберем теперь алгоритм, который избавляется от общего выбора в терме $P \,[\!]\, Q$, имея ввиду закон

$$P \,[\!]\, Q = Q \,[\!]\, P$$

ОбщийВыбор$(P \,[\!]\, Q)$
если $P = (x_1 : B_1 \to R_1)$ **и** $Q = (x_2 : B_2 \to R_2)$ **то**
применяем **обобщенный** закон
конец
если $P = Q$ **то**
применяем закон $P \,[\!]\, P = P$
конец
если $Q =$ STOP **или** $P =$ STOP **то**
применяем закон $P \,[\!]\,$ STOP $=$ STOP
конец
если $Q =$ SKIP **или** $P =$ SKIP **то**
применяем закон $P \,[\!]\,$ SKIP $= P \sqcap$ SKIP
конец

если $Q = (R \sqcap S)$ **или** $P = (R \sqcap S)$ **то**
применяем закон $P \,[\!]\, (R \sqcap S) = (P \,[\!]\, R) \sqcap (P \,[\!]\, S)$
считая для определенности, что $Q = (R \sqcap S)$.
ОбщийВыбор$(P \,[\!]\, R)$
ОбщийВыбор$(P \,[\!]\, S)$
конец
если $Q =$ (IF G THEN R ELSE S) **или** $P =$ (IF G THEN R ELSE S) **то**
применяем закон
$P \,[\!]\,$ IF G THEN R ELSE $S =$ IF G THEN $(P \,[\!]\, R)$ ELSE $(P \,[\!]\, S)$,
считая для определенности, что $Q =$ (IF G THEN R ELSE S).
ОбщийВыбор$(P \,[\!]\, R)$
ОбщийВыбор$(P \,[\!]\, S)$
конец
если $Q = (R \,[\!]\, S)$ **или** $P = (R \,[\!]\, S)$ **то**
ОбщийВыбор$(R \,[\!]\, S)$
Считаем для определенности, что $Q = (R \,[\!]\, S)$.
После того, как общий выбор в подтерме Q разобран, Q будет
преобразован в некоторый Q', и нужно разобрать общий выбор
в терме $P \,[\!]\, Q'$:
ОбщийВыбор$(P \,[\!]\, Q')$
конец
если $Q = (R;\ S)$ **или** $P = (R;\ S)$ **то**
считая для определенности, что $Q = (R;\ S)$, применяем
РазборСеквенциальногоОператора$(R;\ S)$,
который преобразует Q в некоторый Q'; а затем
ОбщийВыбор$(P \,[\!]\, Q')$
конец
если $Q = (x := expr;\ R)$ **или** $P = (x := expr;\ R)$ **то**
присваивание „приклеивается “к терму P (алгоритм склеивания будет
приведен при разборе интерпретации присваивания), и вызываются
ОбщийВыбор$(P \,[\!]\, R)$ или
ОбщийВыбор$(R \,[\!]\, Q)$ соответственно,
конец
если $Q = (R \parallel S)$ **или** $P = (R \parallel S)$ **то**
запустим отдельный алгоритм, разбирающий эту ситуацию
ПараллельностьВОбщемВыборе$(P \,[\!]\, Q)$
Такой алгоритм должен проносить параллельность внутрь термов R
и S и вызывать **ОбщийВыбор**, пока не дойдет до множественного
выбора. Далее терм разбирается в соответствии с **обобщенным**
законом.
конец

Заметим, что в процессе разбора термов P и Q мы всегда найдем множественный выбор (кроме тех случаев, когда P или Q есть STOP или SKIP , а тогда имеются соответствующие законы), поскольку рекурсивному вызову процесса, по условию, наложенному тулом, должно предшествовать хотя бы одно событие.

5.7 Секвенциальный оператор

Рассмотрим процесс обработки выражения вида $(P;\ Q)$, встретившегося в *ProcessDescription* некоторого процесса U.

РазборСеквенциального оператора$(P;\ Q)$
Сначала „разберем “первый операнд.
если $P =$ STOP **то конец**
согласно закону STOP ; $Q =$ STOP
если $P =$ SKIP **то конец**
согласно закону SKIP ; $P =$ SKIP
если $P = (R;\ S)$ **то**
РазборСеквенциального оператора$(R;\ (S;\ Q))$
конец
согласно закону $(P;\ Q);\ R = P;\ (Q;\ R)$
если $P = (e : B \to R(e))$ **то**
РазборСеквенциального оператора$(R(e);\ Q)$ для всех $e \in B$
конец

согласно закону $(e : B \to R(e));\ Q = (e : B \to R(e);\ Q)$
если $P =$ (IF G THEN R ELSE R) **то**
РазборСеквенциального оператора$(R;\ Q)$
РазборСеквенциального оператора$(S;\ Q)$
конец
согласно закону
(IF G THEN R ELSE R); $Q =$ IF G THEN $(R;\ Q)$ ELSE $(S;\ Q)$
если $P = (R \sqcap S)$ **или** $P = (R \between S)$ **то**
РазборСеквенциального оператора$(R;\ Q)$
РазборСеквенциального оператора$(S;\ Q)$
конец
согласно законам
$(R \sqcap S);\ Q = (R;\ Q) \sqcap (S;\ Q)$
$(R \between S);\ R = (R;\ Q) \between (S;\ Q)$
если $P = (R \parallel S)$ **то**
добавим в *ProcessDescription* процесса U строку

$$I = Q,$$

где I – новый процессный идентификатор, являющийся также
состоянием процесса U и проведем
ПараллельнаяКомпозиция$(R \parallel S)$
конец
если $P = (x := e;\ R)$ **то**
приклеиваем присваивание к R, а затем применяем
РазборСеквенциального оператора$(R;\ Q)$
конец
Выделим также особый случай
если $Q =$ SKIP **то конец**
согласно закону $P;$ SKIP $= P$
Осталось рассмотреть последний случай при разборе P
если $P = J$, где J – идентификатор процесса **то**

заменим в данном его вхождении выражение $(J;\ Q)$ на J, добавим в *ProcessDescription* процесса U строку

$$I = Q,$$

здесь I – новый идентификатор процесса, который является новым состоянием машины. Далее нужно занятся разбором процессного терма Q; и в случае появления внутри него секвенциального оператора, вновь применить **РазборСеквенциального оператора**. Если же Q не содержит вхождений секвенциального оператора, то можно начинать последнюю стадию обработки.
Рассмотрим процесс P_J, задаваемый идентификатором J (этот идентификатор также является состоянием машины). Займемся разбором терма P_J, фиксируя при этом состояния, из которых машина переходит в SKIP , назовем их $S_1 \dots S_n$. При этом если SKIP префиксируется событием a и (или) сопровождается условием G

$$a \to (\text{ IF } G \text{ THEN SKIP }),$$

то это выражение заменяется на

$$a \to S_j,$$

где S_j – новое состояние машины; а в операцию B-машины *Success* добавляется ветвь

$$\text{SELECT } G \wedge S_M = S_j \text{ THEN } S_M := I \text{ END },$$

где S_M – переменная состояния машины. Если переход в SKIP не сопровождается условным оператором, то в *Success* добавляется ветвь

$$\text{SELECT } S_M = s_k \text{ THEN } S_M := I \text{ END}$$

Различные ветви, соответствующие состояниям s_i, соединяются операцией выбора [] .
конец

По своей сути событие *Success* является общим для всех последовательных процессов, описываемых в спецификации. Поэтому в случае, когда секвенциальный оператор используется для описания композиции чередующихся процессов, например

PROCESS $S_P = |||v : T.P[v]$ WHERE
$P[v] = a \to$ SKIP ; S

END

то *Success* следует параметризовать индексом чередования v. В случае существования еще одной композиции чередующихся процессов с индексом v', *Success* следует параметризовать и им тоже: $Success[v, v']$. При этом в предусловии *Success* необходимо предусмотреть возможность принятия индексами специального значения $NULL$, аналогично ситуации с *Update*:

$Success(v, v') \mathrel{\widehat{=}}$
PRE $v \in V \cup \{NULL\} \wedge v' \in V' \cup \{NULL\}$
SELECT $v \neq NULL \wedge SP(v) = I$ THEN $SP(v) = J$ END
SELECT $v' \neq NULL \wedge SQ(v') = I'$ THEN $SQ(v') = J'$ END
...
END

Таким образом, каждая из ветвей SELECT будет менять состояние только одного процесса из одной композиции чередования.
Рассмотрим еще одну особеннность последовательных процессов. Пусть R – некоторый нераспараллеливающийся процесс, S – процессный идентификатор последовательного процесса, к которому R рекурсивно обращается. Проблема состоит в том, что обращений может быть несколько, и после успешного завершения своей работы R может переходить в различные состояния, например

$$R = d \rightarrow S;\ a \rightarrow \text{SKIP};\ S;\ b \rightarrow R$$

$$S = c \rightarrow \text{SKIP}$$

Для того, чтобы S совершал после окончания работы правильные переходы, необходимо завести переменную $Valid_{R,S}$, которая будет указывать S, в какое состояние следует перейти. Областью значений $Valid_{R,S}$, естественно, будет являться множество состояний процесса R. Рассмотрим конкретный рекурсивный вызов S внутри R:

$$\ldots e \rightarrow S;\ K$$

Пусть процессный терм K при разборе секвенциального оператора заменен на процессный идентификатор S_K, тогда присваивание $Valid_{R,S} := S_K$ следует внести в ветвь SELECT события e, соответствующую данному переходу по идентификатору S:

$$\text{SELECT } G \wedge S_R = I \text{ THEN } S_R, Valid_{R,S} := S, S_K \text{ END}$$

В ветвях операции *Success*, соответствующих переходам в SKIP внутри процесса S следует совершать переходы по значению $Valid_{R,S}$:

$$\text{SELECT } G \wedge S_R = I \text{ THEN } S_R := Valid_{R,S} \text{ END}$$

5.8 Присваивание

Рассмотрим процессный терм вида $(x := e;\ P)$, где - список переменных, – список B-выражений, – процессный терм. Присваивание вида

$$x_1, \ldots, x_n := e_1, \ldots, e_n$$

рассматривается как обычная параллельная подстановка в AMN (т.е. сначала вычисляются значения выражений $e_1 \ldots e_n$, а затем производится одновременное присваивание).
Утверждение. *К параллельной подстановке можно привести любую последовательность присваиваний вида*

$$x_1 := e_1;\ \ldots;\ x_n := e_n$$

Доказательство. Проведем индукцию по n. Для $n = 1$ присваивание уже является параллельной подстановкой. Рассмотрим случай

$$x_1, \ldots, x_i := e_1, \ldots e_i;\ x_{i+1} := e_{i+1}(x_1 \ldots x_i);\ \ldots;\ x_n := e_n$$

Если e_{i+1} зависит от $x_1, \ldots x_i$, то заменим в выражении e_{i+1} вхождения $x_k, k = 1 \ldots i$ на e_k в соответствии с законом

$$(x := e;\ y := f(x)) = (x, y := e, f(e))$$

и получим эквивалентное присваивание

$$x_1, \ldots, x_i, x_{i+1} := e_1, \ldots e_i, e_{i+1}(e_1 \ldots e_i);\ \ldots;\ x_n := e_n$$

Таким образом в csp-тексте будут соединяться в параллельную подстановку все идущие подряд присваивания. Рассмотрим теперь алгоритм „приклеивания “присваивания $x := e$ к процессному терму P.

Присваивание$(x := e;\ P)$
если $P =$ STOP **то конец**
согласно закону $(x := e;\ \text{STOP}) = \text{STOP}$
если $P = (R;\ S)$ **то**
Присваивание$(x := e;\ R)$
конец
согласно закону $(x := e;\ (R;\ S)) = (x := e;\ R);\ S$
если $P = (R \parallel S)$ **или** $P = (R \sqcap S)$ **или** $P = (R \between S)$ **или** $P = (R \,[]\, S)$ **или** $P = (R \setminus C)$ **то**
Присваивание$(x := e;\ R)$
Присваивание$(x := e;\ S)$
конец
согласно законам
$(x := e;\ (P\ op\ S)) = (x := e;\ P)\ op\ (x := e;\ R)$,
где $op \in \{\parallel, \sqcap, \between, [], \setminus\}$
если $P =$ (IF $G(x)$ THEN R ELSE S) **то**
заменим $G(x)$ на $G(e)$ в условном операторе
Присваивание$(x := e;\ R)$
Присваивание$(x := e;\ S)$
конец
согласно закону
$(x := e;$ (IF $G(x)$ THEN R ELSE $S)) =$
IF $G(e)$ THEN $(x := e;\ R)$ ELSE $(x := e;\ S)$
если $P =$ SKIP **то**
изменим ветвь SELECT операции *Success*, соответствующую данному переходу в SKIP
SELECT $G \wedge S_M = I$ THEN $S_M, y := J, f(x)$ END ,
на следующую:
SELECT $G \wedge S_M = I$ THEN $x, S_M, y := e, J, f(e)$ END
если $P = (a \to J)$ **то**
изменим ветвь SELECT , соответствующую данному переходу в состояние I
SELECT $G \wedge S_M = I$ THEN $S_M, y := J, f(x)$ END ,
на следующую:
SELECT $G \wedge S_M = I$ THEN $x, S_M, y := e, J, f(e)$ END
конец
если P – процессный идентификатор **то**
данное вхождение P в процессный терм префиксировано вхождением некоторого события a, изменим
ветвь SELECT операции a, соответствующую данному переходу в состояние P
SELECT $G \wedge S_M = I$ THEN $S_M := P$ END ,
на следующую:
SELECT $G \wedge S_M = I$ THEN $S_M, x := P, e$ END
конец

5.9 Недетерминированный выбор

Пусть в процессе разбора csp-текста мы нашли терм R вида $(P \sqcap Q)$ в *ProcessDescription* некоторого процесса U. Для того, чтобы обработать $(P \sqcap Q)$, нужно найти событие a, префиксирующее наш недетерминированный выбор и вставить в тело операции, соответствующей a, разрешение недетерминизма. При этом необходимо ввести следующее понятие

Определение. Рассмотрим некоторое вхождение операции *op* в описание некоторого процесса. Пусть терм P является одним из операндов *op*. Тогда *op* называется *внешней операцией по отношению к данному вхождению терма P в csp-текст*, или, короче, *внешней операцией P*, если из контекста ясно, о каком вхождении идет речь.

Заметим, что ситуации, когда внешней операцией по отношению к недетерминизму являются присваивание, секвенциальный оператор или общий выбор уже рассмотрены в предыдущих разделах, т.е. приведены соответствующие законы преобразования процессных термов.

Рассмотрим теперь алгоритм нахождения префиксирующего события для недетерминированного выбора.

НедетерминированныйВыбор$(P \sqcap Q)$
если внешняя операция $(P \sqcap Q)$ есть параллельность $S \parallel (P \sqcap Q)$ **то**
НедетерминированныйВыбор$((S \parallel P) \sqcap (S \parallel Q))$

конец
согласно закону $S \parallel (P \sqcap Q) = (S \parallel P) \sqcap (S \parallel Q)$
если внешняя операция $(P \sqcap Q)$ есть условный оператор
IF G THEN $(P \sqcap Q)$ **то**
НедетерминированныйВыбор((IF G THEN P) $\sqcap$ (IF G THEN Q))
конец
согласно закону IF G THEN $(P \sqcap Q) =$ (IF G THEN P) $\sqcap$ (IF G THEN Q)
если внешняя операция $(P \sqcap Q)$ есть недетерминированный выбор
$S \sqcap (P \sqcap Q)$ **то**
рассматриваем такой терм как недетерминированный выбор трех процессов
НедетерминированныйВыбор$(S \sqcap P \sqcap Q)$
конец
согласно закону $S \sqcap (P \sqcap Q) = (S \sqcap P) \sqcap Q$
если внешняя операция $(P \sqcap Q)$ есть префиксирование $a \rightarrow (P \sqcap Q)$ **то**
искомое префиксирующее событие есть a
конец
если внешней операции нет, т.е. рассматриваемое вхождение недетерминированного выбора образует строку $R = (P \sqcap Q)$ **то**
рассмотрим все рекурсивные вызовы по идентификатору R в *ProcessDescription* процесса U и для каждого такого вызова
если вызов находится внутри *ProcessDescription* **то**
этот вызов префиксирован некоторым событием a, которое является искомым
конец
если вызов расположен после слова PROCESS **то**
префиксирующего события нет
конец

Пусть префиксирующее событие a существует, I – состояние, из которого процесс U переходит в $(P_1 \sqcap \ldots \sqcap P_n)$ через событие a. Добавим в *ProcessDescription* процесса, где находится обрабатываемое вхождение недетерминизма, следующие строки:

$$S_{P_1} = P_1$$
$$\ldots$$
$$S_{P_n} = P_n$$

При этом в множество состояний машины добавятся новые состояния $S_{P_1} \ldots S_{P_n}$. В операцию, соответствующую событию a, добавим следующую ветвь SELECT :

SELECT $U = I$ THEN
ANY s WHERE $s \in \{S_{P_1} \ldots S_{P_n}\}$ THEN $U := s$ END
END

Рассмотрим теперь ситуацию, когда префиксирующего события не существует, т.е. недетерминизм находится на внешнем уровне. Пусть также U не является чередованием нескольких индексированных процессов.

PROCESS $U = R$ WHERE
$R = (P_1 \sqcap \ldots \sqcap P_n)$
ProcessDescription
END

В этом случае строка

$$R = (P_1 \sqcap \ldots \sqcap P_n)$$

заменяется на строки

$$S_{P_1} = P_1$$
$$\ldots$$
$$S_{P_n} = P_n,$$

идентификаторы $S_{P_1} \ldots S_{P_n}$ добавляются в множество состояний, а разрешение недетерминизма вставляется в качестве одной из параллельных подстановок в раздел INITIALISATION B-машины:

$$\text{ANY } s \text{ WHERE } s \in \{S_{P_1} \ldots S_{P_n}\} \text{ THEN } U := s \text{ END}$$

Пусть теперь U является чередованием индексированного множества процессов $R[v]$.

PROCESS $U = |||v : V.R[v]$ WHERE
$R[v] = (P_1[v] \sqcap \ldots \sqcap P_n[v])$
ProcessDescription
END

Различие в интерпретации недетерминизма по сравнению с предыдущим случаем состоит лишь в более сложной инициализации, связанной с интерпретацией чередования: пространство состояний процесса U задается функцией из множества индексов V в множество процессных идентификаторов, определенных в *ProcessDescription* процесса U.

ANY s WHERE $s \in \{S_{P_1} \ldots S_{P_n}\}$ THEN
$U := \lambda\, v.(v \in V \mid s)$
END

5.10 Параллельная композиция с синхронизацией по множеству событий

Пусть в процессе разбора csp-текста мы нашли в *ProcessDescription* некоторого процесса U терм R вида $P \mid \{e_1, \ldots e_n\} \mid Q$. Специальной обработки такого терма не требуется, если $P =$ STOP или $Q =$ STOP . Терм R при этом вырождается в STOP согласно закону

$$P \parallel \text{STOP} = \text{STOP}$$

Для того, чтобы обработать $P \mid \{e_1, \ldots e_n\} \mid Q$ в общем случае, нужно найти событие a, префиксирующее параллельную композицию и вставить в тело операции, соответствующей a инициализацию состояний параллельных процессов P и Q. Заметим, что ситуации, когда внешней операцией по отношению к параллельной композиции являются присваивание, секвенциальный оператор или общий выбор уже рассмотрены в предыдущих разделах, т.е. приведены соответствующие законы преобразования процессных термов. В случае, если внешней операцией по отношению к параллельной компоизции является недетерминизм, нужно найти подстановку ANY , в которой данное вхождение недетерминированного выбора разрешается, и вставить инициализацию состояний P и Q туда.
Вообще, для описания параллельной композиции процесов $(P_1 \parallel \ldots \parallel P_n)$ нужно знать множества событий $C_i, i = 1 \ldots n$, по которым происходит синхронизация: процесс P_i синхронизируется с $(P_1 \parallel \ldots \parallel P_{i-1} \parallel P_{i+1} \parallel \ldots P_n)$ по событиям из C_i. Для случая расширенного синтаксиса

$Q =\parallel (P_1, \ldots P_n)$ WHERE
PROCESS $P_1 = R_1$ CONSTRAINS $e_i \ldots e_j$ WHERE
ProcessDescriptions
...
PROCESS $P_n = R_n$ CONSTRAINS $e_k \ldots e_l$ WHERE
ProcessDescriptions
END

множества C_i указаны в части CONSTRAINS определения процессов P_i. Для композиции двух процессов $P \mid \{e_1, \ldots e_n\} \mid Q$, $C_P = C_Q = \{e_1, \ldots e_n\}$. При добавлении к бинарной композиции еще одного параллельного процесса $(P \mid \{e_1, \ldots e_n\} \mid Q) \mid \{e_{n+1}, \ldots e_m\} \mid R$ происходит расширение множества синхронизации $C_P = C_Q = C_R = \{e_1, \ldots e_m\}$. Таким образом, оба синтаксических вида параллельной композиции можно рассматривать унифицированно.
Рассмотрим теперь алгоритм нахождения префиксирующего события либо подходящей подстановки ANY для параллельной композиции.

ПараллельнаяКомпозиция$(P \parallel Q)$
если внешняя операция $(P \parallel Q)$ есть недетерминированный выбор
$S \sqcap (P \parallel Q)$ **то**
НедетерминированныйВыбор$(S \sqcap (P \parallel Q))$
при этом найдем подстановку ANY , в которой разрешается
недетерминизм
конец
если внешняя операция $(P \parallel Q)$ есть условный оператор
IF G THEN $(P \parallel Q)$ **то**
ПараллельнаяКомпозиция((IF G THEN P) $\parallel$ (IF G THEN Q))
конец
согласно закону
IF G THEN $(P \parallel Q) =$ (IF G THEN P) $\parallel$ (IF G THEN Q)
если внешняя операция $(P \parallel Q)$ есть параллельная композиция

$S \parallel (P \parallel Q)$ **то**
рассматриваем такой терм как параллельную композицию трех процессов
ПараллельнаяКомпозиция$(S \parallel P \parallel Q)$
конец
согласно закону $S \parallel (P \parallel Q) = (S \parallel P) \parallel Q$
если внешняя операция $(P \parallel Q)$ есть префиксирование $a \to (P \parallel Q)$ **то**
искомое префиксирующее событие есть a
конец
если внешней операции нет, т.е. рассматриваемое вхождение параллельной композиции образует строку $R = (P_1 \parallel \ldots \parallel P_n)$ **то**
рассмотрим все рекурсивные вызовы по идентификатору I в *ProcessDescription* процесса U и для каждого такого вызова
если вызов находится внутри *ProcessDescription* **то**
этот вызов префиксирован некоторым событием a, которое является искомым
конец
если вызов расположен после слова PROCESS **то**
префиксирующего события нет
конец

Пусть префиксирующее событие a существует, I – состояние, из которого процесс U переходит в $(P_1 \parallel \ldots \parallel P_n)$ через событие a. Поскольку U разветвляется, одной переменной S_U, описывающей его состояние, недостаточно. Заведем по переменной S_{P_i}, которая будет указывать на текущее состояние процесса P_i, и множеству $State_{P_i}$, которое является пространством состояний P_i, для каждого из процессов P_i в параллельной композиции. У каждого из P_i должны быть особое состояние *NotActivated*, указывающее, разделился ли U на параллельные ветви. Такое же состояние появляется и у U – при разветвлении таким образом будет указываться, что процесс U уснул вплоть до соединения своих ветвей. В раздел INITIALISATION B-машины следует вставить подстановку

$$S_{P_1}, \ldots, S_{P_n} := \mathit{NotActivated}, \ldots, \mathit{NotActivated}$$

а в операцию события a – ветвь SELECT

$$\text{SELECT } S_U = I \text{ THEN } S_{P_1}, \ldots, S_{P_n} := P_1, \ldots, P_n \text{ END}$$

где P_i являются состояниями соответствующих параллельных процессов при расширенном синтаксисе описания параллельной композиции. Если же параллельность встречается в виде

$$P \mid \{e_1, \ldots e_n\} \mid Q$$

внутри терма, то нужно, вставив в *ProcessDescription* процесса U строки

$$P_{init} = P$$

$$Q_{init} = Q$$

инициализировать параллельную композицию в операции префиксирующего события a ветвью SELECT

$$\text{SELECT } S_U = I \text{ THEN } S_U, S_P, S_Q := \mathit{NotActivated}, P_{init}, Q_{init} \text{ END}$$

После разделения на ветви U может слиться в один процесс единственным образом

$$(P_1 \parallel \ldots \parallel P_n);\ K$$

При разборе секвенциального оператора определится состояние I_K, в которое должны перейти процесс U после успешного завершения всех P_i. Поэтому для каждого P_i следует найти все состояния I_{P_i,k_i} из которых P_i переходит в SKIP , соединить ветви SELECT

$$\text{SELECT } S_{P_i} = I_{P_i,k_i} \text{ THEN } S_{P_i}, S_U := \mathit{NotActivated}, I_K \text{ END}$$

B-операцией выбора [] в подстановку $Subst_{P_i}$, а затем добавить подстановку

$$Subst_{P_1} \parallel \ldots \parallel Subst_{P_n}$$

в тело операции *Success*. Таким образом будет обеспечено слияние процессов $P_1 \ldots P_n$ в один.
За синхронизацией следует проследить при определении тел операций, соответствующих событиям из множеств C_i. Для каждого состояния $e \in C_k \cap \ldots \cap C_l$ находим все пары состояний $(I'_{e,r,q}, I''_{e,r,q}), q = 1 \ldots v$; что в процессе P_r есть переход $I'_{e,r,q} \xrightarrow{e} I''_{e,r,q}$. Создадим для каждого P_r ветвь SELECT

SELECT $S_{P_r} = I'_{e,r,1}$ THEN $S_{P_r} := I''_{e,r,1}$ END
[] ... []
SELECT $S_{P_r} = I'_{e,r,v}$ THEN $S_{P_r} := I''_{e,r,v}$ END

и все такие ветви для $P_k, \ldots, P_l$ соединим в параллельную подстановку B-операцией $\|$. Результирующую подстановку вставим в операцию события e.
Пусть теперь алгоритм **ПараллельнаяКомпозиция** выдал не событие, а подстановку ANY

$$\text{ANY } s \text{ WHERE } s \in \{S_{Q_1} \ldots S_{Q_m}\} \text{ THEN } S_U := s \text{ END}$$

где Q_i и есть наша параллельная композиция $(P_1 \parallel \ldots \parallel P_n)$. В этом случае инициализацию переменных состояния параллельных ветвей нужно вставить в подстановку ANY :

ANY s WHERE $s \in \{S_{Q_1} \ldots S_{Q_m}\}$ THEN
SELECT $s = S_{Q_i}$ THEN
$S_{P_1}, \ldots, S_{P_n} := P_1, \ldots, P_n$
ELSE $S_U := s$
END
END

Если префиксирующего события нет и параллельность не входит в чередование

PROCESS $S_U = U$ WHERE
$U = (P_1 \parallel \ldots \parallel P_n)$ WHERE
PROCESS $P_1[v] = T_1$ WHERE
ProcessDescription
END
...
PROCESS $P_n[v] = T_n$ WHERE
ProcessDescription
END
END

то инициализация переменных состояния параллельных ветвей

$$S_{P_1}, \ldots, S_{P_n} := P_1, \ldots, P_n$$

вставляется непосредственно в раздел INITIALISATION B-машины.
Если все же U является чередованием индексированного множества процессов $R[v]$

PROCESS $U = |||v : V.R[v]$ WHERE
$R[v] = (P_1[v] \parallel \ldots \parallel P_n[v])$ WHERE
PROCESS $P_1[v] = T_1$ WHERE
ProcessDescription
END
...
PROCESS $P_n[v] = T_n$ WHERE
ProcessDescription
END
END

то инициализация переменных состояния параллельных ветвей в разделе B-машины INITIALISATION будет выглядеть следующим образом:

$$S_{P_1}, \ldots, S_{P_n} := \lambda\, v.(v \in V \mid P_1), \ldots, \lambda\, v.(v \in V \mid P_n)$$

В этом случае переменная U, отвечающая за состояние асинхронной композиции в целом, не нужна: ее заменяют переменные $S_{P_1}, \ldots, S_{P_n}$.

5.11 Проверка предваренности рекурсии

Важным ограничением csp2B является требование предваренности всякого рекурсивного вызова процесса. Это делается для того, чтобы корректное описание процесса приводилось к нормальной форме, на основании которой легко сформировать пространство состояний процесса и все ветви SELECT для различных

событий. Это разумное требование сохраняется и при рассмотрении дополнительных операций над процессами: после их интерпретации с параллельным применением стандартных законов преобразования процессных термов из csp2B описание процесса автоматически приведется к нормальной форме. Синтаксически непредваренным может быть только рекурсивный вызов во втором аргументе секвенциального оператора $(P;\ I)$, здесь P – процессный терм, а I – идентификатор; поскольку семантически такой вызов предварен событием *Success*.
Проблема состоит в том, что предваренность рекурсии может нарушаться при использовании операции сокрытия, а поэтому необходимо, параллельно с обработкой процессных термов, проверять корректность рекурсивного вызова. Рассмотрим алгоритм который проверяет предваренность рекурсии в произвольном процессном терме P. Алгоритм, кроме самого терма P, получает в качестве аргумента флаг f, принимающий значения *true* и *false*, который говорит о том, предварен или нет терм P, а также множество событий C, которые подлежат сокрытию в терме P.

ПредваренностьРекурсии(P, f, C)
если $P =$ SKIP **или** $P =$ STOP **то конец**
если $P = (R \sqcap S)$ **или** $P = (R \parallel S)$ **или** $P = (R \,[\![\, S)$ **или** $P = (x := e;\ S)$ **или**
$P = \mu\, X \bullet S(X)$ **или** $P = (R \,[]\, S)$ **или** $P =$ IF G THEN R ELSE S **то**
ПредваренностьРекурсии(R, false, C)
ПредваренностьРекурсии(S, false, C)
конец
если $P = (R \setminus C')$ **то**
ПредваренностьРекурсии$(R, f, C \cup C')$
конец
если $P = (R;\ S)$ **то**
ПредваренностьРекурсии(R, f, C)
ПредваренностьРекурсии(S, true, C)
конец
если $P = (x : B \rightarrow R(x))$ **то**
для каждого $x \in B$
если $x \in C$ **то**
ПредваренностьРекурсии$(R(x), f, C)$
конец
если $x \notin C$ **то**
ПредваренностьРекурсии$(R(x), \mathit{true}, C)$
конец
частными случаями множественного выбора являются также
префиксирование и временное префиксирование
если $P = (\ \textsc{Wait}\ t;\ R)$ **то**
ПредваренностьРекурсии(R, true, C)
конец
если $P = (e \rightarrow R) \rhd S$ **или** $P = (e \rightarrow R) \rhd \{t\} S$ **то**
если $e \in C$ **то**
ПредваренностьРекурсии(R, f, C)
ПредваренностьРекурсии(S, f, C)
конец
если $x \notin C$ **то**
ПредваренностьРекурсии(R, true, C)
ПредваренностьРекурсии(S, f, C)
конец
если $P = I$ где I – процессный идентификатор **то**
если $f = \mathit{true}$ **то** рекурсия предварена
если $f = \mathit{false}$ **то** рекурсия непредварена
конец

5.12 Взаимодействие между процессами

Рассмотрим пару параллельных процессов P и Q, синхронизированных по каналу c, который переносит сообщения типа T. Пусть процессы описаны в *ProcessDescription* некоторого процесса U. Конечно, P и Q могут быть описаны как параллельные и внешним образом:

PROCESS $S_P = P$ WHERE
ProcessDescription(P)
END

PROCESS $S_Q = Q$ WHERE
ProcessDescription(Q)
END

В этом случае в качестве *ProcessDescription* рассматриваются описания процессов P и Q. Пусть для определенности сообщения по каналу c передаются от P к Q. Определим в процессе разбора *ProcessDescription* все пары состояний процесса P, $(I'_{P,i}, I''_{P,i}), i = 1 \dots n$, что некоторое событие $c!t$ переводит процесс P из состояния $I'_{P,i}$ в состояние $I''_{P,i}$; а также аналогичные пары $(I'_{Q,j}, I''_{Q,j}), j = 1 \dots m$ для процесса Q. Прием сообщений с канала c может встречаются в определении процесса Q следующим образом:

$$c?x \rightarrow R(x)$$

При этом переменная x, индексирующая процесс R, становится переменной B-машины. Для каждой пары $(I'_{Q,j}, I''_{Q,j})$ определим переменную x_j, в которой хранится индекс процесса, соответствующий данному переходу. Тело операции, соответствующей каналу c, выглядит следующим образом :

$c(p) \mathrel{\widehat{=}}$
PRE $p \in T$ THEN
(SELECT $S_P = I'_{P,1}$ THEN $S_P := I''_{P,1}$ END
[] ... []
SELECT $S_P = I'_{P,n}$ THEN $S_P := I''_{P,n}$ END)
||
(SELECT $S_Q = I'_{P,1}$ THEN $x_1, S_Q := p, I''_{P,i}$ END
[] ... []
SELECT $S_Q = I'_{P,m}$ THEN $x_n, S_Q := p, I''_{P,m}$ END)
END

Таким образом, сообщение, посылаемое процессом P, представляется в виде параметра p операции c. Другой способ приема сообщений

$$c.x \rightarrow R$$

где R – идентификатор процесса. В этом случае не заводится переменной, хранящей x, а в ветвь SELECT , соответствующую данному переходу, вставляется предусловие:

$$\text{SELECT } S_P = I \wedge p = x \text{ THEN } S_P := R \text{ END}$$

где p – аргумент операции c. В случае, если x является индексом чередования, то никакой дополнительной обработки не требуется: операция c будет фактически параметризована индексом x.

6 Уточнение спецификаций

При проектировании сложных информационных систем из компонентов совершенно необходимым становится понятие уточнения спецификаций. В процессе конструирования системы происходит конкретизация ее спецификации, причем основным моментом является выражение операций и структур данных исходной спецификации в терминах ресурсов. Важно, чтобы факт уточнения при этом был доказуемым.

Инструментом для конструирования спецификаций из компонентов является B-технология, которая применима также и к AMN, расширенной средствами управления процессами. Тул, работающий на основе алгоритмов, приведенных в разделе (5), должен переводить абстрактное описание процесса в терминах расширенной нотации в стандартную B-машину. При этом события из алфавита процесса становятся операциями B-машины. Множества, переменные, инвариант, используемые при описании процесса, как и в стандартной B-машине, определены определены в рамках типизированной теории множеств, а операции, полученные в результате трансляции, являются обобщенными подстановками [3]. Описание процесса может включать в качестве событий операции B-машин, указанных в секции CONJOINS . Таким образом будет использована функциональность готовой B-машины, и при этом наложены ограничения на порядок применения операций, согласно логике процесса.

Для B-машины, полученной после трансляции процессного описания, может быть доказана ее непротиворечивость, заключающаяся в существовании состояния переменных машины, при котором удовлетворяется инвариант, а также в сохранении инварианта после инициализации и применения любой из операций машины. Непротиворечивость машины подтвердит возможность ее использования: операции можно будет применять только в соответствии с логикой процесса.

Для непротиворечевого описания процесса можно написать спецификацию, являющуюся его уточнением – REFINEMENT. В уточнении происходит конкретизация течения процесса: возможно расширение алфавита, появление событий, детализирующих процесс. События, из исходного описания процесса могут быть реализованы с использованием функциональности некоторых B-машин.

Отношение между переменными B-машины, полученной после трансляции описания процесса, и переменными B-машины уточнения описывается в инварианте уточнения. Вообще, при уточнении CSP-машин

(описания процессов) необходимо сначала конвертировать CSP-машину и ее уточнение в B-машины. Таким образом определится множество состояний как процесса, так и его уточнения, а также набор возможных переходов между состояниями. После этого и нужно написать инвариант для уточнения, который бы задавал отношение между пространствами состояний машин [5], добавить этот инвариант в CSP-уточнение, а затем вновь произвести конвертацию в B-машину. Далее можно проводить доказательство корректности уточнения, пользуясь B-Toolkit. B-Toolkit сгенерирует теоремы, доказательство которых будет гарантировать существование состояния переменных, удовлетворяющего инвариантам машины и ее уточнения, а также выполнение инварианта после инициализации и применения любой из операций уточнения.

Последним этапом конструирования системы является написание IMPLEMENTATION — окнчательного уточнения, в котором нет собственных переменных, параллельных подстановок, недетерминизма и предусловий в операциях. Все структуры данных и функциональность в имплементации заимствуются из импортируемых машин. Таким образом, система оказывается построенной из компонентов — импортируемых машин и факт корректности уточнения ею исходной спецификации будет доказанным, в случае выполнения всех теорем, сгенерированных B-Toolkit в течении всего процесса уточнения.

7 Пример спецификации ситемы

Для демонстрации возможностей расширенной нотации AMN приведем пример системы лифтов: часто встречающаяся на практике, такая система тем не менее достаточно сложна, чтобы показать преимущества совмещения AMN и процессного подхода, заимствованного из CSP. В модели широко используются последовательные взаимодействующие через каналы процессы, отмечены аспекты, связанные с работой системы в реальном времени.
Система разделена на две машины – B-машину *LSData*, содержащую сложные структуры данных и операции по работе с ними, а также csp-машину *LiftSystem*, задающую процессную логику работы. Операции B-машины используются csp-машиной в качестве событий, поэтому *LSData* и присоединяется к *LiftSystem* с помощью CONJOINS .
Лифты, общим количеством *Width*, расположены в здании высотой в *Height* этажей. На каждом из этажей расположена пара кнопок, с помощью которых люди могут вызывать лифт для следования в требуемом направлении – *UpButton* и *DownButton*. Запросы кладутся в очередь, а затем назначаются свободным лифтам. Каждый лифт содержит панель с кнопками, каждая из которых сответствует этажу. Внутренний запрос лифта после нажатия кнопки регистрируется контроллером лифта. При остановке на этаже дверь открывается, а затем, после некоторого ожидания закрывается. Закрытие может быть прервано пассажиром, прохождение через дверь которого фиксируется некоторыми сенсорами.

MACHINE *LSData*
SETS
$ButtonStatus = \{On, Off\}$;
$MoveDirection = \{Up, Down\}$;
$DoorMess = \{ToOpen, ToClose, Opened, Closed, Interrupt\}$

Множество *ButtonStatus* отражает состояния кнопки, которая может быть включена – *On* – человеком и выключена – *Off* – системой. *MoveDirection* задает направления движения лифта, а *DoorMess* – сообщения, с помощью которых дверь лифта общается со своими внутренними механизмами.

CONSTANTS *Height*, *Width*
PROPERTIES $Height \in \mathbb{N} \wedge Width \in \mathbb{N}$
VARIABLES
Panels,
irs, *ups*, *dns*,
reqQ, *req*, *Cups*, *Cdns*

В разделе внутренних переменных

- *Panels* – кнопки, находящихся внутри лифтов;
- *irs* – нажатые внутренние кнопки, это множество может быть разделено на
- *ups*, *dns* – запросы на этажи выше и ниже данного соответственно;
- *reqQ* – очередь запросов с этажей;
- *req* – множество запрашиваемых этажей;
- *Cups*, *Cdns* – множество запросов с этажей между двумя данными на лифт вверх и лифт вниз соответственно.

INVARIANT
$Panels \in \text{seq}(\text{seq}(ButtonStatus)) \wedge$
$\text{size}(Panels) = Width \wedge$
$\forall i.(i \in 1..Width \Rightarrow \text{size}(Panels(i)) = Height) \wedge$
$irs \in \text{seq}(\mathbb{P}(\mathbb{N})) \wedge \text{size}(irs) = Width \wedge$
$\forall i.(i \in 1..Width \Rightarrow irs(i) = \{f \mid f \in 1..Height \wedge Panels(i)(f) = On\}) \wedge$
$ups \in \text{seq}(\mathbb{N} \nrightarrow \mathbb{P}(\mathbb{N})) \wedge \text{size}(ups) = Width \wedge$
$\forall i.(i \in 1..Width \Rightarrow ups(i) = \lambda fl.(fl \in 1..Height \mid \{n \mid n \in irs(i) \wedge n \geqslant fl\}))$
$\wedge\ dns \in \text{seq}(\mathbb{N} \nrightarrow \mathbb{P}(\mathbb{N})) \wedge \text{size}(dns) = Width \wedge$
$\forall i.(i \in 1..Width \Rightarrow dns(i) = \lambda fl.(fl \in 1..Height \mid \{n \mid n \in irs(i) \wedge n \leqslant fl\}))$
$\wedge$
$reqQ \in seq(1..Height \times MoveDirection) \wedge req \in 1..Height \leftrightarrow MoveDirection \wedge$
$Cups \in 1..Height \times 1..Height \nrightarrow \mathbb{P}(1..Height) \wedge$
$Cdns \in 1..Height \times 1..Height \nrightarrow \mathbb{P}(1..Height) \wedge$
$req = \text{ran}(reqQ) \wedge$
$\forall(f_1, f_2).(f_1 \in 1..Height \wedge f_2 \in 1..Height \Rightarrow$

$Cups(f_1, f_2) = \{f \mid f \in 1..Height \wedge f \mapsto Up \in req \wedge f_1 \leq f \wedge f \leq f_2\} \wedge$
$Cdns(f_1, f_2) = \{f \mid f \in 1..Height \wedge f \mapsto Down \in req \wedge f_1 \leq f \wedge f \leq f_2\})$

INITIALISATION
ANY f WHERE
$f \in \text{seq}(\text{seq}\, ButtonStatus) \wedge \text{size}(f) = Width \wedge$
$\forall (i, j).(i \in 1..Height \wedge j \in 1..Width \Rightarrow \text{size}(f(j)) = Height \wedge f(j)(i) = Off)$
THEN $Panels := f$
END
||
ANY f WHERE
$f \in \text{seq}(\mathbb{P}(\mathbb{N})) \wedge \text{size}(f) = Width \wedge$
$\forall i.(i \in 1..Width \Rightarrow f(i) = \varnothing)$
THEN $irs := f$
END
||
ANY f WHERE
$f \in \text{seq}(\mathbb{N} \nrightarrow \mathbb{P}(\mathbb{N})) \wedge \text{size}(f) = Width \wedge$
$\forall i.(i \in 1..Width \Rightarrow f(i) = \varnothing)$
THEN $ups := f$
END
||
ANY f WHERE
$f \in \text{seq}(\mathbb{N} \nrightarrow \mathbb{P}(\mathbb{N})) \wedge \text{size}(f) = Width \wedge$
$\forall i.(i \in 1..Width \Rightarrow f(i) = \varnothing)$
THEN $dns := f$
END
||
$reqQ := \varnothing \parallel req := \varnothing \parallel Cups := \varnothing \parallel Cdns := \varnothing$

OPERATIONS
$fl' \leftarrow Next(l, fl, md) \mathrel{\widehat{=}}$
PRE $l \in 1..Width \wedge fl \in 1..Height \wedge md \in MoveDirection$
THEN
IF $md = Up$ THEN
IF $ups(l)(fl) \neq \varnothing$ THEN $fl' := \min(ups(l)(fl))$
ELSE IF $dns(l)(fl) \neq \varnothing$ THEN $fl' := \max(dns(l)(fl))$
END
ELSE
IF $dns(l)(fl) \neq \varnothing$ THEN $fl' := \max(dns(l)(fl))$
ELSE IF $ups(l)(fl) \neq \varnothing$ THEN $fl' := \min(ups(l)(fl))$
END
END

С помощью *Next* внутренняя очередь лифта определяет следующий этаж, на котором должен остановится лифт.

$PanelsOn\_Act(l, fl) \mathrel{\widehat{=}}$
PRE $l \in 1..Width \wedge fl \in 1..Height$
THEN
$Panels(l)(fl) := On \quad \parallel$
$irs(l) := irs(l) \cup \{fl\} \parallel$
ANY f WHERE
$f \in \mathbb{N} \nrightarrow \mathbb{P}(\mathbb{N}) \wedge$
$f = \lambda fl.(fl \in 1..Height \mid \{n \mid n \in irs(l) \cup \{fl\} \wedge n \geqslant fl\})$
THEN $ups(l) := f$
END ||
ANY f WHERE
$f \in \mathbb{N} \nrightarrow \mathbb{P}(\mathbb{N}) \wedge$
$f = \lambda fl.(fl \in 1..Height \mid \{n \mid n \in irs(l) \cup \{fl\} \wedge n \leqslant fl\})$
THEN $dns(l) := f$
END
END

PanelsOn включает кнопку лифта при внутреннем запросе, а также изменяет внутренние переменные системы с учетом нового запроса.

$PanelsOff\_Act(l, fl) \mathrel{\widehat{=}}$
PRE $l \in 1..Width \land fl \in 1..Height$
THEN
$Panels(l)(fl) := Off \quad \|$
$irs(l) := irs(l) - \{fl\} \|$
ANY f WHERE
$f \in \mathbb{N} \nrightarrow \mathbb{P}(\mathbb{N}) \land$
$f = \lambda fl.(fl \in 1..Height \mid \{n \mid n \in irs(l) - \{fl\} \land n \geqslant fl\})$
THEN $ups(l) := f$
END $\|$
ANY f WHERE
$f \in \mathbb{N} \nrightarrow \mathbb{P}(\mathbb{N}) \land$
$f = \lambda fl.(fl \in 1..Height \mid \{n \mid n \in irs(l) - \{fl\} \land n \leqslant fl\})$
THEN $dns(l) := f$
END
END

PanelsOff гасит кнопку лифта при приезде на соответствующий этаж, а также изменяет внутренние переменные с учетом выполненного запроса.

$RemoveReq\_Act(pair)$
PRE $pair \in 1..Height \mapsto MoveDirection$
THEN
$reqQ := reqQ \rhd \{pair\} \|$
$req := \mathrm{ran}\,(reqQ \rhd \{pair\}) \|$
ANY f WHERE
$f \in 1..Height \times 1..Height \nrightarrow \mathbb{P}(1..Height) \land$
$\forall(f_1, f_2).(f_1 \in 1..Height \land f_2 \in 1..Height \Rightarrow$
$f(f_1, f_2) = \{g \mid g \in 1..Height \land$
$g \mapsto Up \in \mathrm{ran}\,(reqQ \rhd \{pair\}) \land f_1 \leq g \land g \leq f_2\})$
THEN $Cups := f$
END $\|$
ANY f WHERE
$f \in 1..Height \times 1..Height \nrightarrow \mathbb{P}(1..Height) \land$
$\forall(f_1, f_2).(f_1 \in 1..Height \land f_2 \in 1..Height \Rightarrow$
$f(f_1, f_2) = \{g \mid g \in 1..Height \land$
$g \mapsto Down \in \mathrm{ran}\,(reqQ \rhd \{pair\}) \land f_1 \leq g \land g \leq f_2\})$
THEN $Cdns := f$
END $\|$
END

RemoveReq удаляет из очереди внешних запросов назначенный запрос.

$AddReq\_Act(pair)$
PRE $pair \in 1..Height \mapsto MoveDirection$
THEN
$reqQ := reqQ^\frown < pair > \|$
$req := \mathrm{ran}\,(reqQ^\frown < pair >) \|$
ANY f WHERE
$f \in 1..Height \times 1..Height \nrightarrow \mathbb{P}(1..Height) \land$
$\forall(f_1, f_2).(f_1 \in 1..Height \land f_2 \in 1..Height \Rightarrow$
$f(f_1, f_2) = \{g \mid g \in 1..Height \land$
$g \mapsto Up \in \mathrm{ran}\,(reqQ^\frown < pair >) \land f_1 \leq g \land g \leq f_2\})$
THEN $Cups := f$
END $\|$
ANY f WHERE
$f \in 1..Height \times 1..Height \nrightarrow \mathbb{P}(1..Height) \land$
$\forall(f_1, f_2).(f_1 \in 1..Height \land f_2 \in 1..Height \Rightarrow$
$f(f_1, f_2) = \{g \mid g \in 1..Height \land$
$g \mapsto Down \in \mathrm{ran}\,(reqQ^\frown < pair >) \land f_1 \leq g \land g \leq f_2\})$
THEN $Cdns := f$

END
END

AddReq добавляет к очереди внешних запросов новый запрос.

END .

MACHINE *LiftSystem*
CONJOINS *LSData*, *Time*
CONSTANTS
$t_o, t_p,$
GetFirst, *GetSecond*

Временные константы t_o и t_p определяют время ожидания пассажиров перед закрытием двери и время ожидания внутреннего запроса в лифте соответственно. Функции *GetFirst* и *GetSecond* выделяют из запроса этаж и направление соответственно.

PROPERTIES
$t_o \in \mathbb{T} \wedge t_p \in \mathbb{T} \wedge$
$GetFirst \in 1..Height \times MoveDirection \rightarrow 1..Height \wedge$
$GetSecond \in 1..Height \times MoveDirection \rightarrow MoveDirection \wedge$
$\forall(fl, md).(fl \in 1..Height \wedge md \in MoveDirection \Rightarrow$
$\quad GetFirst(fl \mapsto md) = fl \wedge GetSecond(fl \mapsto md) = md)$
VARIABLES
Floors,
LCfl, *LCmd*

В разделе внутренних переменных

- *Floors* – кнопки на этажах;
- *LCfl* – этажи, на которых находятся лифты;
- *LCmd* – текущие направления движения лифтов.

INVARIANT
$Floors \in \text{seq}(\text{seq}(ButtonStatus)) \wedge$
$\text{size}(Floors) = Height \wedge$
$\forall i.(i \in 1..Height \Rightarrow \text{size}(Floors(i)) = 2) \wedge$
$\wedge\ LCfl \in \text{seq}(1..Height) \wedge LCmd \in \text{seq}(MoveDirection) \wedge$
$\text{size}(LCfl) = Width \wedge \text{size}(LCmd) = Width$

DEFINITIONS
$UpButton(n) == \text{first}(Floors(n));$
$DownButton(n) == \text{last}(Floors(n))$

INITIALISATION
ANY f WHERE
$f \in \text{seq}(\text{seq}\, ButtonStatus) \wedge \text{size}(f) = Height \wedge$
$\forall i.(i \in 1..Height \Rightarrow \text{size}(f(i)) = 2 \wedge$
$\quad \text{first}(f(i)) = Off \wedge \text{last}(f(i)) = Off)$
THEN $Floors := f$
END
||
ANY f WHERE
$f \in \text{seq}(1..Height) \wedge \text{size}(f) = Width \wedge$
$\forall i.(i \in 1..Width \Rightarrow f(i) = 1)$
THEN $LCfl := f$
END
||
ANY f WHERE
$f \in \text{seq}(MoveDirection) \wedge \text{size}(f) = Width \wedge$
$\forall i.(i \in 1..Width \Rightarrow f(i) = Up)$
THEN $LCmd := f$
END

CHANNELS
$Enter(n : 1..Height, md : MoveDirection)$
$Service(n : 1..Height, md : MoveDirection)$
$Move(l : 1..Width, n : 1..Height - 1)$
$Open(l : 1..Width) \quad Close(l : 1..Width) \quad Conf(l : 1..Width)$
$Arrive(l : 1..Width)$
$IntShed(l : 1..Width, fl : 1..Height, md : MoveDirection)$
$IntDest(l : 1..Width, dest : 1..Height)$
$IntServ(l : 1..Width, fl : 1..Height)$
$Select(dest : 1..Width, md : MoveDirection)$
$Check(l : 1..Width, fl : 1..Height, dest : 1..Height, md : MoveDirection)$
$CheckDest(l : 1..Width, dest : 1..Height)$
$Checking(l : 1..Width)$

ALPHABET
$Request(n : 1..Height, md : MoveDirection)$
$(l, dm) \leftarrow Servo(lift : 1..Width)$
$Sensor(l : 1..Width, dm : DoorMessage)$
$IntRequest(l : 1..Width, fl : 1..Height)$

PROCESS $Building = |||n : 1..Height.Floor[n]$
CONSTRAINS $(n, md) \leftarrow Enter \quad Service(n, md)$
WHERE
$Floor[n] = \mu\, X \bullet (Request.n.Down \rightarrow PressDown[n]\ []$
$Request.n.Up \rightarrow PressUp[n]\ []$
$Service.n.Down \rightarrow DownOff[n]\ []$
$Service.n.Up \rightarrow UpOff[n]);\ X$
$PressDown[n] =$ IF $DownButton(n) = Off$ THEN
$DownButton[n] := On;\ Enter!n!Down \rightarrow$ SKIP ELSE SKIP
$PressUp[n] =$ IF $UpButton(n) = Off$ THEN
$UpButton := On;\ Enter!n!Up \rightarrow$ SKIP ELSE SKIP
$DownOff[n] = DownButton(n) := Off;$ SKIP
$UpOff[n] = UpButton(n) := Off;$ SKIP
END

Процесс *Building* представляет собой асинхронную параллельную композицию процессов *Floor*. *Floor* принимает запросы по каналу *Request*, зажигая соответствующие запросу кнопки, а затем отправляет эти запросы по каналу *Enter* для дальнейшего распределения. В случае подтверждения выполнения запроса по каналу *Service*, соответствующая кнопка гасится.
Процесс *Lifts* представляет собой асинхронную параллельную композицию процессов *Lift* – лифтов. Каждый лифт есть параллельная композиция процессов *LiftControl*, *Shaft*, *Door*, *InternalQ* – контроллера лифта, шахты, двери и внутренней очереди запросов.

PROCESS $Lifts = |||l : 1..Width.Lift[l]$
CONSTRAINS
$(n, md) \leftarrow Service \quad Select(dest, md) \quad (l, fl, dest, md) \leftarrow Check$
$CheckDest(l, dest) \quad Checking(l) \quad Update$
WHERE
$Lift[l] =\| (SLiftControl[l], SShaft[l], SDoor[l], SInternalQ[l])$ WHERE
PROCESS $SDoor[l] = Door[l]$
CONSTRAINS $Open \quad Close \quad Conf$ WHERE
$Door[l] = \mu\, D[l] \bullet Open.l \rightarrow CycleDoor[l];\ Close!l \rightarrow D[l]$
$CycleDoor[l] = Servo!l!ToOpen \rightarrow OpenDoor[l];\ Conf!l \rightarrow$
$(\mu\, CD[l] \bullet$ WAIT $t_o;\ CloseDoor[l] \triangledown (Sensor.l.Interrupt \rightarrow$
$Servo!l!ToOpen \rightarrow OpenDoor[l];\ CD[l]))$
$OpenDoor[l] = Sensor.l.Opened \rightarrow$ SKIP
$CloseDoor[l] = Servo.l!l!ToClose \rightarrow Sensor.l.Closed \rightarrow$ SKIP
END

Процесс *Door* управляет работой двери, открывая и закрывая ее. Закрытие может быть прервано при получении по каналу *Sensor* сообщения *Interrupt*, которое означает, что человек пересек обозреваемое сенсорами двери пространство. Сервомеханизмам по каналу *Servo* подаются сообщения *ToOpen*, *ToClose*, приказывающие открыть или закрыть дверь. Сенсоры также сообщают процессу, в каком состоянии находится дверь –

Opened либо *Closed*. После открытия двери контроллеру посылается подтверждение о выполнении запроса по каналу *Conf*.

PROCESS $SShaft[l] = Shaft[l]$
CONSTRAINS $Move(l, n) \quad l \leftarrow Arrive$ WHERE
$Shaft[l] = \mu\, S[l] \bullet Move.l?n \rightarrow Arrive!l \rightarrow S[l]$
END

Шахта получает рапоряжения по каналу *Move* – на сколько этажей нужно передвинуть лифт, а затем по каналу *Arrive* уведомляет о прибытии.

PROCESS $SInternalQ[l] = InternalQ[l]$
CONSTRAINS $IntShed(l, fl, md) \quad IntServ(l, fl) \quad (l, fl) \leftarrow IntDest$ WHERE
$InternalQ[l] = \mu\, IQ[l] \bullet$
$IntRequest.l?fl \rightarrow PanelsOn.l.fl \rightarrow IQ[l]\ []$
$IntServ.l?fl \rightarrow PanelsOff.l.fl \rightarrow IQ[l]\ []$
IF $irs(l) \neq \varnothing$ THEN $IntShed.l?fl?md \rightarrow Destination[l](Next(l, fl, md))$
$Destination[l](IQdest : 1..Height) = IntDest!l!IQdest \rightarrow IQ[l]$
END

Внутренняя очередь регистрирует внутренние запросы, поступающие по каналу *IntRequest*, а также подтверждения выполнения запросов по каналу *IntServ*. В случае нахождения лифта в режиме выполнения внутреннего запроса и непустоты *irs*, по каналу *IntShed* может быть получено текущее положение лифта, вычислено место следующей остановки и отправлено по каналу *IntDest*.

PROCESS $SLiftControl[l] = LiftControl[l]$
CONSTRAINS
$l \leftarrow Open \quad Close(l) \quad Conf(l)$
$(l, dest) \leftarrow Move \quad Arrive(l)$
$(l, fl, md) \leftarrow IntShed \quad (l, fl) \leftarrow IntServ \quad IntDest(l, dest)$
WHERE
$LiftControl[l] = \mu\, LC[l] \bullet (IntShed!l!LCfl(l)!LCmd(l) \rightarrow$
$GetInternal[l]\ []\ (\text{Wait}\ t_p;\ GetExternal[l]));\ LC[l]$
$GetInternal[l] = IntDest.l?dest \rightarrow Check!l!LCfl(l)!dest!LCmd(l) \rightarrow$
$(Checking.l \rightarrow Internal[l]\ []\ Check.l?dest \rightarrow Move[l];\ External[l])$
$GetExternal[l] = Select?dest?dir \rightarrow LCmd(l) := dir;\ Move[l];\ External[l]$
$External[l] = Service!LCfl(l)!md \rightarrow Close(l) \rightarrow$ SKIP
$Internal[l] =$
(IF $dest > LCfl(l)$ THEN $LCmd(l) := Up;$ SKIP
ELSE IF $dest < LCfl(l)$ THEN $LCmd(l) := Down;$ SKIP
ELSE SKIP);
$Move[l];\ IntServ!l!LCfl(l) \rightarrow Close.l \rightarrow$ SKIP
$Move[l] =$
IF $dest = LCfl(l)$ THEN $Open!l \rightarrow Conf.l \rightarrow$ SKIP
ELSE $Move!l!(dest - LCfl(l)) \rightarrow Arrive.l \rightarrow Open!l \rightarrow Conf.l \rightarrow$
$LCfl(l) := dest;$ SKIP
END
END
END

Контроллер лифта управляет поведением лифта, т.е. осуществляет взаимодействие между окружением и другими частями лифта, реализуя три режима поведения.
Процесс начинается с ожидания запросов со стороны пассажиров либо центрального контроллера. Пассажирам предоставляется возможность в течение времени t_p сделать внутренний запрос, перед тем, как лифт начнет выполнять внешние запросы.
Если очередь готова к тому, чтобы назначить внутренний запрос, то ей посылается текущее положение лифта по каналу *IntShed*, а затем по каналу *IntDest* принимается пункт назначения. Затем центральный контроллер проверяет, нет ли внешнего запроса с промежуточных этажей. Если есть, то вначале выполняется внешний запрос. В противном случае вычисляется направление движения и движение осуществляется. Затем дверь открывается, о выполнении запроса уведомляются центральный контроллер и внутренняя очередь. После закрытия двери лифт снова переходит в режим ожидания.
Если в течение времени t_p не происходит внутренних запросов, то происходит обслуживание внешнего запроса в соответствии с информацией, полученной по каналу *Select*: устанавливается новое направление движения, происходит движение, дверь открывается. Далее по каналу *Service* отправляется уведомление о

выполнении запроса, и после того, как дверь закрывается, лифт возвращается в режим ожидания.

PROCESS $Controller = Control$
CONSTRAINS
$Enter(n, md) \quad (dest, md) \leftarrow Select \quad Check(l, fl, dest, md)$
$(l, dest) \leftarrow CheckDest \quad l \leftarrow Checking$
WHERE
$Control = \mu\, C \bullet ($
$Enter?ReqDest?ReqMd \rightarrow Join(ReqDest, ReqMd)\ []$
IF $reqQ \neq <>$ THEN $Select!GetFirst(\text{first}\,(reqQ))!GetSecond(\text{first}\,(reqQ)) \rightarrow$
$RemoveReq.\text{first}\,(reqQ) \rightarrow$ SKIP []
$Check?l?fl?d?md \rightarrow Find(l, fl, d, md) \quad);\ C$
$Join(ReqDest : 1..Height, ReqMd : MoveDirection) =$
$AddReq. < ReqDest \mapsto ReqMd > \rightarrow$ SKIP
$Find(l : 1..Width, fl : 1..Height, d : 1..Height, md : MoveDirection) =$
IF $md = Up \wedge Cups(fl, d) \neq \varnothing$ THEN
$CheckDest!l!\min\,(Cups(fl, d)) \rightarrow$
$RemoveReq(\min\,(Cups(fl, d) \mapsto md) \rightarrow$ SKIP
ELSE
IF $md = Down \wedge Cdns(fl, d) \neq \varnothing$ THEN
$CheckDest!l!\min\,(Cdns(fl, d)) \rightarrow$
$RemoveReq(\min\,(Cdns(fl, d) \mapsto md) \rightarrow$ SKIP
ELSE $Cheking!l \rightarrow$ SKIP
END

Задача центрального контроллера состоит в распределении внешних запросов по свободным лифтам. Запросы поступают по каналу *Enter*, после чего кладутся в очередь *reqQ*. Распределение запросов происходит по каналам *Select* и *CheckDest*, при наличии свободных или попутных лифтов соответственно. Перенаправленный запрос удаляется из очереди операцией *RemoveReq*.

END .

Рассмотрим теперь B-машину, в которую тул должен первести csp-машину *LiftSystem*. Содержимое разделов CONSTANTS, PROPERTIES, VARIABLES, INVARIANT, DEFINITIONS, INITIALISATION переносится в B-машину без изменений, поэтому опишем ниже только добавления к этим разделам, связанные с переводом.

MACHINE $LiftSystem$
INCLUDES $LSData$, $Time$
SETS
$NULLSet = \{NULL\}$;
$STimeState = \{Time\}$;
$BuildingState = \{Floor, PressDown, PressUp, DownOff, UpOff, PressDown_1, PressDown_2, PressUp_1, PressUp_2\}$;
$SDoorState = \{NotActivated, Door, Door_1, CycleDoor, CycleDoor_1, CD, OpenDoor, OpenDoor_1, WaitOpen, WaitOpen_1\}$;
$SCloseDoorState = \{NotActivated, CloseDoor, CloseDoor_1, CloseDoor_2\}$;
$SInterruptState = \{NotActivated, SensorInterrupt, SensorInterrupt_1, OpenDoor, OpenDoor_1\}$;
$SShaftState = \{Shaft, Shaft_1\}$;
$SInternalQState = \{InternalQ, InternalQ_1, InternalQ_2, Destination\}$;
$SLiftControlState = \{LiftControl, GetInternal, GetExternal, WaitInput, WaitInput_1, GetInternal_1, GetInternal_2, Internal, Move, External, External_1, External_2, Move_1, Move_2, Move_3, Move_4, Move_5, Move_6, Internal_1, Internal_2, Internal_3\}$;
$ControllerState = \{Control, Control_1, Control_2, Join, Join_1, Find, Find_1, Find_2, Find_3, Find_4, Find_5\}$
VARIABLES
$STime$,
$Building$, Bmd,
$ReqDest$, $ReqMd$,
$SDoor$, $ValidDoorOpenDoor$, $TOpen$, $SCloseDoor$, $SInterrupt$,
$SShaft$,
$SInternalQ$, $IQfloor$, $IQmd$, $IQdest$,
$SLiftControl$, $LCdest$, $TInput$, $ValidLiftControlMove$
$Controller$, Cl, Cfl, Cd, Cmd
INVARIANT
$STime \in STimeState \wedge$
$Building \in 1..Height \rightarrow BuildingState \wedge$
$Bmd \in MoveDirection \wedge$
$ReqDest \in 1..Height \wedge ReqMd \in MoveDirection \wedge$
$SDoor \in 1..Width \rightarrow SDoorState \wedge$
$TOpen \in \mathbb{T} \wedge$
$ValidDoorOpenDoor \in 1..Width \rightarrow SDoorState \wedge$
$SCloseDoor \in 1..Width \rightarrow SCloseDoorState \wedge$
$SInterrupt \in 1..Width \rightarrow SInterruptState \wedge$
$SShaft \in 1..Width \rightarrow SShaftState \wedge$
$MoveDestination \in 1..Width \rightarrow 1..Height - 1 \wedge$
$SInternalQ \in 1..Width \rightarrow SInternalQState \wedge$
$IQfloor \in 1..Width \rightarrow 1..Height \wedge$
$IQmd \in 1..Width \rightarrow MoveDirection \wedge$
$IQdest \in 1..Height \wedge$
$SLiftControl \in 1..Width \rightarrow SLiftControlState \wedge$
$LSdest \in 1..Width \rightarrow 1..Height \wedge$
$TInput \in \mathbb{T} \wedge$
$ValidLiftControlMove \in 1..Width \rightarrow SLiftControlState \wedge$
$Controller \in ControllerState \wedge$
$Cl \in 1..Width \wedge Cfl \in 1..Height \wedge$
$Cd \in 1..Width \wedge Cmd \in MoveDirection$
INITIALISATION
$STime := Time \parallel$
$Building := \lambda fl.(fl \in 1..Height \mid Floor) \parallel$
$SDoor := \lambda l.(l \in 1..Width \mid Door) \parallel$
$SCloseDoor := \lambda l.(l \in 1..Width \mid NotActivated) \parallel$
$SInterrupt := \lambda l.(l \in 1..Width \mid NotActivated) \parallel$
$SShaft := \lambda l.(l \in 1..Width \mid Shaft) \parallel$

$SInternalQ := \lambda\, l.(l \in 1..Width \mid InternalQ) \parallel$
$SLiftControl := \lambda\, l.(l \in 1..Width \mid LiftControl) \parallel$
$Controller := Control$

OPERATIONS
$Update(l) \mathrel{\widehat{=}}$
PRE $l \in 1..Width$
SELECT $STime = Time$ THEN SKIP END $\parallel$
SELECT $STime = Time$ THEN $Update\_Act$ END $\parallel$
(SELECT $SDoor(l) = CD$ THEN
$TOpen := now \parallel SDoor(l) := WaitOpen$
END []
SELECT $SDoor(l) = WaitOpen \wedge now \mapsto add(TOpen, t_o) \in before$
THEN
$SDoor(l) := WaitOpen$
END []
SELECT $SDoor(l) = WaitOpen \wedge \neg\, (now \mapsto add(TOpen, t_o) \in before)$
THEN
$SDoor(l) := WaitOpen_1$
END []
SELECT $SLiftControl(l) = LiftControl$ THEN
$TInput := now \parallel SLiftControl(l) := WaitInput$
END []
SELECT $SLiftControl(l) = WaitInput \wedge$
$now \mapsto add(TInput, t_p) \in before$ THEN
$SLiftControl(l) := WaitInput$
END []
SELECT $SLiftControl(l) = WaitInput \wedge$
$\neg\, (now \mapsto add(TInput, t_p) \in before)$
THEN
$SLiftControl(l) := WaitInput_1$
END)
END

$Success(n, l) \mathrel{\widehat{=}}$
PRE $n \in 1..Height \cup NULLSet \wedge l \in 1..Width \cup NULLSet$
SELECT $n \neq NULL \wedge Building(n) = PressDown_1$ THEN
$Building(n) := Floor$
END []
SELECT $n \neq NULL \wedge Building(n) = PressDown_2$ THEN
$Building(n) := Floor$
END []
SELECT $n \neq NULL \wedge Building(n) = PressUp_1$ THEN
$Building(n) := Floor$
END []
SELECT $n \neq NULL \wedge Building(n) = PressUp_2$ THEN
$Building(n) := Floor$
END []
SELECT $n \neq NULL \wedge Building(n) = DownOff$ THEN
$Building(n) := Floor \parallel DownButton(n) := Off$
END []
SELECT $n \neq NULL \wedge Building(n) = UpOff$ THEN
$Building(n) := Floor \parallel UpButton(n) := Off$
END []
SELECT $l \neq NULL \wedge SDoor(l) := OpenDoor_1$ THEN
$SDoor(l) := ValidDoorOpenDoor$
END []
SELECT $l \neq NULL \wedge SInterrupt(l) := OpenDoor_1$ THEN
$SDoor(l) := ValidDoorOpenDoor \parallel SInterrupt := NotActivated$
END []
SELECT $l \neq NULL \wedge SDoor(l) = WaitOpen_1$ THEN
$SDoor(l) := NotActivated \parallel SCloseDoor(l) := CloseDoor \parallel$
$SInterrupt(l) := SensorInterrupt$
END []

SELECT $l \neq NULL \wedge SCloseDoor(l) = CloseDoor_2$ THEN
$SDoor(l) := Door_1 \parallel SCloseDoor(l) := NotActivated \parallel$
$SInterrupt(l) := NotActivated$
END []
SELECT $l \neq NULL \wedge SLiftControl(l) = WaitInput_1$ THEN
$SLiftControl(l) = GetExternal$
END []
SELECT $l \neq NULL \wedge SLiftControl(l) = External_2$ THEN
$SLiftControl(l) := LiftControl$
END []
SELECT $l \neq NULL \wedge SLiftControl(l) = Internal \wedge$
$LCdest(l) > LCfl(l)$ THEN
$SLiftControl(l) := Move \parallel LCmd(l) := Up \parallel$
$ValidLiftControlMove(l) := Internal_1$
END []
SELECT $l \neq NULL \wedge SLiftControl(l) = Internal \wedge$
$\neg (LCdest(l) > LCdest(l)) \wedge LCdest(l) < LCfl(l)$ THEN
$SLiftControl(l) := Move \parallel LCmd(l) := Down \parallel$
$ValidLiftControlMove(l) := Internal_1$
END []
SELECT $l \neq NULL \wedge SLiftControl(l) = Internal \wedge$
$\neg (LCdest(l) > LCfl(l)) \wedge \neg (LCdest(l) < LCfl(l))$ THEN
$SLiftControl(l) := Move \parallel ValidLiftControlMove(l) := Internal_1$
END []
SELECT $l \neq NULL \wedge SLiftControl(l) = Move_5$ THEN
$SLiftControl(l) := ValidLiftControlMove(l) \parallel LCfl(l) := LCdest(l)$
END []
SELECT $l \neq NULL \wedge SLiftControl(l) = Move_6$ THEN
$SLiftControl(l) : - = ValidLiftControlMove(l)$
END
SELECT $l \neq NULL \wedge SLiftControl(l) = Internal_3$ THEN
$SLiftControl(l) := LiftControl$
END []
SELECT $Controller = Control_2$ THEN
$Controller := Control$
END []
SELECT $Controller = Join_1$ THEN
$Controller := Control$
END []
SELECT $Controller = Find_2$ THEN
$Controller := Control$
END []
SELECT $Controller = Find_4$ THEN
$Controller := Control$
END []
SELECT $Controller = Find_5$ THEN
$Controller := Control$
END
END

$Request(n, md) \mathrel{\widehat{=}}$
PRE $n \in 1..Height \wedge md \in MoveDirection$ THEN
SELECT $Building(n) = Floor \wedge md = Down$ THEN
$Building(n) := PressDown$
END []
SELECT $Building(n) = Floor \wedge md = Up$ THEN
$Building(n) := PressUp$
END
END

$Service(n, md) \mathrel{\widehat{=}}$
PRE $n \in 1..Height \wedge md \in MoveDirection$ THEN
(SELECT $Building(n) = Floor \wedge md = Down$ THEN
$Building(n) := DownOff$

```
        END []
        SELECT Building(n) = Floor ∧ md = Up THEN
          Building(n) := UpOff
        END )
      ||
      ( ANY l WHERE
          l ∈ 1..Width THEN
          SELECT SLiftControl[l] = External THEN
            SLiftControl(l) := External_1 || Bmd := md
          END
        END )
    END

Enter(n, md) ≙
    PRE n ∈ 1..Height ∧ md ∈ MoveDirection THEN
      ( SELECT Building(n) = PressDown ∧ DownButton(n) = Off THEN
          DownButton(n) = On || ReqDest := n || ReqMd := Down ||
          Building(n) := PressDown_1
        END []
        SELECT Building(n) = PressDown ∧ ¬ (DownButton(n) = Off) THEN
          Building(n) := PressDown_2
        END []
        SELECT Building(n) = PressUp ∧ UpButton(n) = Off THEN
          UpButton(n) = On || ReqDest := n || ReqMd := Up ||
          Building(n) := PressUp_1
        END []
        SELECT Building(n) = PressUp ∧ ¬ (UpButton(n) = Off) THEN
          Building(n) := PressUp_2
        END )
      ||
      ( SELECT Controller = Control THEN
          Controller := Join
        END )
    END

Open(l) ≙
    PRE l ∈ 1..Width THEN
      ( SELECT SDoor(l) = Door THEN SDoor(l) := CycleDoor END )
        ||
      ( SELECT SLiftControl(l) = Move ∧ LCdest(l) = LCfl(l) THEN
          SLiftControl(l) := Move_1
        END []
        SELECT SLiftControl(l) = Move_3
          SLiftControl(l) := Move_4
        END )
    END

(l, dm) ← Servo(lift) ≙
    PRE lift ∈ 1..Width THEN
      SELECT SDoor(lift) = CycleDoor THEN
        SDoor(lift) := OpenDoor || l := lift || dm := ToOpen ||
        ValidDoorOpenDoor(lift) := CycleDoor_1
      END []
      SELECT SInterrupt(lift) = SensorInterrupt_1 THEN
        SensorInterrupt(lift) := OpenDoor || l := lift || dm := ToOpen ||
        ValidDoorOpenDoor(lift) := CD
      END []
      SELECT SCloseDoor(lift) = CloseDoor THEN
        SCloseDoor(lift) := CloseDoor_1 || l := lift || dm := ToClose
      END
    END

Sensor(l, dm) ≙
    PRE l ∈ 1..Width ∧ dm ∈ DoorMessage THEN
```

```
      SELECT SDoor(l) = OpenDoor ∧ dm = Opened THEN SDoor := OpenDoor_1 END []
      SELECT SInterrupt(l) = SensorInterrupt ∧ dm = Interrupt THEN
        SCloseDoor(l) := NotActivated || SInterrupt := SensorInterrupt_1
      END []
      SELECT SInterrupt(l) = OpenDoor ∧ dm = Opened THEN
        SInterrupt := OpenDoor_1
      END
      SELECT SCloseDoor(l) = CloseDoor_1 ∧ dm = Closed THEN
        SCloseDoor(l) := CloseDoor_2
      END
    END

Conf(l) ≙
    PRE l ∈ 1..Width THEN
      ( SELECT SDoor = CycleDoor_1 THEN SDoor := CD END )
          ||
      ( SELECT SLiftControl(l) = Move_1 THEN
            SLiftControl(l) := Move_6
        END []
        SELECT SLiftControl(l) = Move_4
          SLiftControl(l) := Move_5
        END )
    END

Close(l) ≙
    PRE l ∈ 1..Width THEN
      ( SELECT SDoor(l) := Door_1 THEN SDoor(l) := Door END )
          ||
      ( SELECT SLiftControl(l) = External_1 THEN
            SLiftControl(l) := External_2
        END []
       SELECT SLiftControl(l) = Internal_2 THEN
            SLiftControl(l) := Internal_3
        END )
    END

Move(l, n) ≙
    PRE l ∈ 1..Width ∧ n ∈ 1..Height − 1 THEN
      ( SELECT SShaft(l) = Shaft THEN SShaft(l) := Shaft_1 END )
          ||
      ( SELECT SLiftControl(l) = Move ∧ ¬ (LCdest(l) = LCfl(l))
        SLiftControl(l) := Move_2 || MoveDestination(l) := LCdest(l) − LCfl(l)
        END )
    END

Arrive(l) ≙
    PRE l ∈ 1..Width THEN
      ( SELECT SShaft(l) = Shaft_1 THEN SShaft(l) := Shaft END )
          ||
      ( SELECT SLiftControl(l) = Move_2 THEN
            SLiftControl(l) := Move_3
        END )
    END

IntRequest(l, fl) ≙
    PRE l ∈ 1..Width ∧ fl ∈ 1..Height THEN
      SELECT SInternalQ(l) = InternalQ THEN
        SInternalQ(l) := InternalQ_1 || IQfloor(l) := fl
      END
    END

PanelsOn(l, fl) ≙
    PRE l ∈ 1..Width THEN
      SELECT SInternalQ(l) = InternalQ_1 ∧ fl = IQfloor(l) THEN
```

$SInternal(l) := InternalQ \parallel PanelsOn\_Act(l, fl)$
END
END

$IntServ(l, fl) \mathrel{\widehat{=}}$
PRE $l \in 1..Width \wedge fl \in 1..Height$ THEN
(SELECT $SInternalQ(l) = InternalQ$ THEN
$SInternalQ(l) := InternalQ_2$
END)
$\parallel$
(SELECT $SLiftControl(l) = Internal_1$ THEN
$SLiftontrol(l) := Internal_2 \parallel IQfloor(l) := LCfl(l)$
END)
END

$PanelsOff(l, fl) \mathrel{\widehat{=}}$
PRE $l \in 1..Width \wedge fl \in 1..Height$ THEN
SELECT $SInternalQ(l) = InternalQ_2 \wedge fl = IQfloor(l)$ THEN
$SInternal(l) := InternalQ \parallel PanelsOff\_Act(l, fl)$
END
END

$RemoveReq(pair) \mathrel{\widehat{=}}$
PRE $pair \in 1..Height \times MoveDirection$ THEN
SELECT $Controller = Control_1$ THEN
$Controller := Control_2 \parallel RemoveReq\_Act(pair)$
END []
SELECT $Controller = Find_1$ THEN
$Controller := Find_2 \parallel RemoveReq\_Act(pair)$
END []
SELECT $Controller = Find_3$ THEN
$Controller := Find_4 \parallel RemoveReq\_Act(pair)$
END
END

$AddReq(pair) \mathrel{\widehat{=}}$
PRE $pair \in 1..Height \times MoveDirection$ THEN
SELECT $Controller = Join$ THEN
$Controller := Join_1 \parallel AddReq\_Act(pair)$
END
END

$IntShed(l, fl, md) \mathrel{\widehat{=}}$
PRE $l \in 1..Width \wedge fl \in 1..Height \wedge md \in MoveDirection$ THEN
(SELECT $SInternalQ(l) = InternalQ \wedge irs(l) \neq \varnothing$ THEN
$SInternalQ(l) := Destination \parallel IQdest := Next(l, IQfloor(l), IQmd(l))$
END)
$\parallel$
(SELECT $SLiftControl(l) = LiftControl$ THEN
$SLiftControl(l) = GetInternal \parallel$
$IQfloor(l) := LCfl(l) \parallel IQmd(l) := LCmd(l)$
END)
END

$IntDest(l, fl) \mathrel{\widehat{=}}$
PRE $l \in 1..Width \wedge fl \in 1..Height$ THEN
(SELECT $SInternalQ(l) = Destination$ THEN
$SInternalQ(l) := InternalQ \parallel LCdest(l) := IQdest$
END)
$\parallel$
(SELECT $SLiftControl(l) = GetInternal$ THEN
$SLiftControl(l) := GetInternal_1$
END)
END

$Check(l, fl, d, md) \mathrel{\widehat{=}}$
PRE $l \in 1..Width \wedge fl \in 1..Height \wedge d \in 1..Height \wedge$
$md \in MoveDirection$ THEN
(SELECT $SLiftControl(l) = GetInternal_1$ THEN
$SLiftControl(l) := GetInternal_2 \parallel$
$Cl := l \parallel Cfl := LCfl(l) \parallel Cd := LCdest \parallel Cmd := LCmd(l)$
END)
$\parallel$
(SELECT $Controller = Control$ THEN
$Controller := Control_3$
END)
END

$Checking(l) \mathrel{\widehat{=}}$
PRE $l \in 1..Width$ THEN
(SELECT $SLiftControl(l) = GetInternal_2$ THEN
$SLiftControl(l) := Internal$
END)
$\parallel$
(SELECT $Controller = Find \wedge \neg (Cmd = Up \wedge Cups(Cfl, Cd) \neq \varnothing) \wedge$
$\neg (Cmd = Down \wedge Cdns(Cfl, Cd) \neq \varnothing)$ THEN
$Controller := Find_5$
END)
END

$CheckDest(l, dest) \mathrel{\widehat{=}}$
PRE $l \in 1..Width \wedge dest \in 1..Width$ THEN
(SELECT $SLiftControl(l) = GetInternal_2$ THEN
$SLiftControl(l) := Move \parallel ValidLiftControlMove(l) := External$
END)
$\parallel$
(SELECT $Controller = Find \wedge Cmd = Up \wedge Cups(Cfl, Cd) \neq \varnothing$ THEN
$Controller := Find_1 \parallel LCdest(l) := dest$
END []
SELECT $Controller = Find \wedge \neg (Cmd = Up \wedge Cups(Cfl, Cd) \neq \varnothing) \wedge$
$md = Down \wedge Cdns(Cfl, Cd) \neq \varnothing$ THEN
$Controller := Find_3 \parallel LCdest(l) := dest$
END)
END

$Select(dest, dir) \mathrel{\widehat{=}}$
PRE $dest \in 1..Height \wedge dir \in MoveDirection$ THEN
ANY l WHERE
$l \in 1..Width$ THEN
SELECT $SLiftControl = GetExternal$ THEN
$SLiftControl(l) := Move \parallel LCmd(l) := dir \parallel$
$ValidLiftControlMove(l) := External$
END
$\parallel$
SELECT $Controller = Control \wedge reqQ \neq <>$ THEN
$Controller := Control_1 \parallel LCdest(l) := dest \parallel LCDir(l) := dir$
END
END
END

END .

8 Синтаксические правила описания процессов

В этом разделе приведены синтаксические правила описания процессов расширенной AMN в форме Бэкуса-Наура. $t \mid s$ означает выбор t или s, $< t >$ означает одно или ни одного вхождения t, $<< t >>$ означает произвольное количество вхождений t.

processes ::= *process* << *process* >>
process ::= *parallel* |
PROCESS *ident* =< |||*identlist* : *explist*. > *ident* < [*identlist*] >< (*explist*) >
< CONSTRAINS *constrainlist* > WHERE
<< *ident* < [*identlist*] < (*typedidentlist*) >= *proc*1 >>
END
parallel ::= || (*recurselist*) WHERE *processes*
*proc*1 ::= *proc*2 << [] *proc*2 >>
| *proc*2 \ *setexpr*
| *proc*2 ⊓ *proc*2
| *proc*2 ⫴ *proc*2
| *proc*2 | {*identlist*} | *proc*2
| *ident* := *exp*; *proc*2
| *proc*2; *proc*3
| *μ ident* *proc*2
| (*ident* : *setexpr* → *proc*2)
| SKIP
| WAIT *ident*; *proc*2
| *event* *ident* → *proc*3
| (*event* → *proc*3) ▷ *proc*2
| (*event* → *proc*3) ▷ {*ident*}*proc*2
| *parallel*
*proc*2 ::= STOP | IF *exp* THEN *proc*2 < ELSE *proc*2 >
| (*proc*1) | *event* → *proc*3
*proc*3 ::= *proc*2 | *recurse*
recurse ::= *ident* < [*identlist*] >< *explist* >
recurselist ::= *recurse* <<, *recurse* >>
event ::= *ident* << .*exp* |?*ident* >><<!*exp* >>
constrainlist ::= *constrain* << *constrain* >>
constrain ::=< *ident* ←| (*identlist*) ←> *ident* < (*identlist*) >
setexpr ::= *ident* | определение множества в B
typedidentlist ::= *ident* : *exp* <<, *ident* : *exp* >>
identlist ::= *ident* <<, *ident* >>
ident ::= идентификатор из букв и цифр, первой стоит буква
explist ::= *exp* <<, *exp* >>
exp ::= B-выражение

9 Заключение

В дипломной работе разработано расширение нотации абстрактных машин средствами описания процессов. Приведены B-конструкции, в которые происходит трансляция процессного описания, в том числе и средства реализации временных возможностей; а также алгоритмы конвертирования описания процессов в стандартную B-машину. Расширенная нотация поддерживает последовательные процессы, параллельность на произвольном уровне, сокрытие, недетерминированный и общий выбор, прерывание, а также временные операции – задержку, таймаут и временное префиксирование. При описании процессов возможно использование выражений, типов и предикатов стандартной AMN, а также использование в качестве событий операций других B-машин. Таким образом достигается интеграция расширенной нотации и стандартной AMN.

Интерпретация операций конструирования процессов позволяет строить уточнения спецификаций процессов и доказывать его корректность с использованием B-технологии. Этим обеспечивается возможность проектирования процессных систем из компонентов.

Расширенная AMN нотация позволяет описывать параллельные системы, взаимодействующие в реальном времени и оперирующие со сложными структурами данных, важным примером которых являются потоки работ.

Возможности нотации продемонстрированы на примере ситемы лифтов, логика работы которой ясно описана в процессной части спецификации.

Список литературы

[1] Л. А. Калиниченко. СИНТЕЗ: Язык определения, проектирования и программирования интероперабельных сред неоднородных информационных ресурсов. М.: ИПИ РАН, 1993.

[2] Ч. Хоар. Взаимодействующие последовательные процессы. М.: Мир, 1989.

[3] J. -R. Abrial. The B-Book. Cambridge University Press, 1996.

[4] B-Toolkit User's Manual - Release 3.4. Copyright B-Core (UK) Ltd., 1997.

[5] M. J. Butler. csp2B: A Practical Approach To Combining CSP and B. Declarative Systems and Software Engineering Group, Technical Report DSSE-TR-99-2, February 1999.

[6] M. J. Butler. csp2B User Manual - Version 1.1. Доступно на http://www.ecs.soton.ac.uk, 1999.

[7] R. Duke, P. King, G. Rose and G.Smith. The Object-Z specification language: Version 1. Technical Report 91-1, Software Verification Research Centre, Australia, 1991

[8] A. Elmagarmid and W. Du. Workflow Management: State of Arts vs. State of Market. Advances in Workflow Management Systems and Interoperability, Istanbul, August 1997.

[9] L. A. Kalinichenko. Workflow Reuse and Semantic Interoperation Issues. Advances in Workflow Management Systems and Interoperability, Istanbul, August 1997.

[10] B. Mahony and J. S. Dong. Timed Communicating Object Z. IEEE Transactions on Software Engineering, Vol. 26 (2), Feb 2000. (ISSN 00985589)

[11] R. Milner. Communication and Concurrency. Prentice Hall, 1989.

[12] S. Schneider. An Operational Semantics for Timed CSP. Information and Computation, 116(2), 1995.

[13] S. Schneider and J. Davies. A Brief History of Timed CSP. Theoretical Computer Science, 138, 1995.

[14] J. Spivey. The Z Notation: A Reference Manual (2nd Ed.). International Series in Computer Science. Prentice Hall, 1992.