View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CU Digital Repository

Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Ondřej Černý

State Space Symmetry Reduction for TBP Analysis

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Ondřej Šerý, Ph.D.

Study programme: Computer Science

Specialization: Depependable Systems

Prague 2011

It is a pleasure to thank my supervisor RNDr. Ondrej Šerý, Ph.D. for his patient guidance, valuable comments and thoughtful ideas. Many thanks are due to RNDr. Tomáš Poch, Ph.D. for sharing his interesting thoughts. Last but not least, I would like to thank my parents and friends for their support and patience.

I declare that I carried out this master thesis independently, and only with cited sources, literature and other professionaly sources.

I understand that my work relates to the rights and obligations under Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact, that the Charles University in Prague has the right to conclude license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, August 4, 2011 Onřej Černý

Title: State Space Symmetry Reduction for TBP Analysis

Author: Ondřej Černý

Department: Depratment of Distributed and Dependable Systems

Supervisor: RNDr. Ondřej Šerý, Ph.D.

Abstract:
Threaded Behavioral Protocols (TBP) is a specification language for modelling the behavior of software components. This thesis aims at an analysis of TBP specifications within environments which involve an unbounded replication of threads. Such a TBP specification – together with a model of the possible environments – induces infinite state space which contains a vast amount of symmetries caused by thread replication. A model checking technique addressing such a state space and reducing the symmetries by using symbolic counter abstraction is proposed. In order to utilize the symbolic counter abstraction, the properties of the TBP specifications (called provisions) are converted into thread state reachability properties. The proposed analysis is safe in the sense that it discovers all errors in the model. On the other hand, it may yield spurious errors, i.e., errors that do not correspond to any real error in the model. The spurious errors are well identified and further possibilities to reduce them are outlined. Beyond the scope of the specific specifications, this work may also present a small step towards supporting dynamic thread creation in TBP.

Keywords: Behavior modelling, component systems, model checking, replication, threads.

Název práce: Redukce symetrií stavového prostoru analýzy TBP

Autor: Ondřej Černý

Katedra: Katerdra Distribuovaných a Spolehlivých Systémů

Vedoucí diplomové práce: RNDr. Ondřej Šerý, Ph.D.

Abstrakt:
Threaded Behavior Protocols (TBP) je specifikační jazyk pro modelování chování softwarových komponent. Tato práce se zaměřuje na analýzu TBP specifikací v rámci prostředí, která obsahují neomezené množšví replikovaných vláken. Takové specifikace spolu s modely možných prostředí způsobí nekonečnost stavového prostoru analýzy, který obsahuje velké množství symetrií, způsobených replikací vláken. V práci je navžena technika analýzy takových modelů, která redukuje symetrie s použitím abstrakce zvané Symbolic Counter Abstraction. Pro její použití je však nutné převést vlastnosti modelu na problém dosažitelnosti stavů vláken. Navrhovaná technika je bezpečná ve smyslu odhalení všech chyb v modelu. Na druhou stranu může způsobovat tzv. *spurious erros*, tj. chyby které neodpovídají skutečným chybám v modelu. Tyto chyby jsou v práci dobře identifikovány a dále jsou nastíněny způsoby jejich redukce. Práce navíc může představovat malý krok směrem k podpoře dynamického vytváření vláken v TBP specifikacích.

Klíčová slova: Modely chování, komponentové systémy, model checking, replikace, vlákna.

Contents

Introduction **3**

Goals . . . 4

Structure of the text . . . 5

Notation . . . 6

1 Symbolic Counter Abstraction **9**

1.1 Vector Addition Systems with States . . . 9

1.2 Karp-Miller Tree . . . 9

1.3 Counter Abstraction . . . 10

2 Threaded Behavior Protocols **13**

2.1 Syntax . . . 13

2.1.1 Type Declarations . . . 14

2.1.2 State Variables . . . 14

2.1.3 Reactions . . . 14

2.1.4 Threads . . . 17

2.1.5 Provisions . . . 17

2.2 Semantics . . . 19

2.2.1 TBP Model . . . 19

2.2.2 Closed Model Analysis . . . 25

2.2.3 Open Model Analysis . . . 28

3 Analysis of Incomplete TBP Model **33**

3.1 Incomplete TBP Model . . . 33

3.1.1 Restricting Provision Expressions . . . 35

3.2 Analysis Using Symbolic Counter Abstraction . . . 37

3.2.1 Artificial Environment from Unbound Provisions . . . 38

3.2.2 Imperative Parts . . . 57

3.2.3 Bound Provisions . . . 66

4 Prototype Implementation **79**

4.1 Structure of the TBP Toolchain . . . 79

4.1.1 TBPLib . . . 79

4.1.2 Badger . . . 79

4.2 Limitations . . . 81

Conclusion **83**

Bibliography **86**

Appendices **87**

A Content of the enclosed CD ROM **87**

Introduction

Component Based Development (CBD) is currently a well established paradigm being used in software development. Its basic idea is to compose complex software from well defined units of software (called components). Individual components are logically isolated and communicate with each other through well defined interface. This isolation allows components to be developed separately or even by different vendors. It also encourages reuse of components in different applications requiring the same functionality.

In theory, a whole new application can be created solely as a composition of third-party prefabricated components. In such a scenario, focus is shifted from the development itself to assuring correctness of a composition of components. Verification of mutual compatibility between individual components becomes a major challenge.

Common component systems such as EJB [Sak09], DCOM [GG97], SOFA 2 [BHP06] or Fractal [BCL$^+$06] are based on component model and provide runtime support for component applications. The component model itself provides means for describing individual components and their communication interfaces and structure of the composition – component architecture. In hierarchical component systems (e.g. SOFA, Fractal) individual components may also be formed as a composition of other components.

However, for the purpose of compatibility verification, the information contained in the component model must be enriched with a description of behavior of the components. There are several theoretical frameworks (e.g. process algebras [Bae05] or interface automata [dAH01]) which may be used to address this issue, but they are too complex to be used as part of the development process. There are also modelling languages focused on software behavior such as Promela [Hol03] but their main disadvantage is an absence of the concept of components.

For this purpose, a component behavior modelling language called Threaded Behavioral Protocols (TBP) was designed. It was introduced in [KPS09] and [Poc10]. TBP aims at the specification of components behavior, whereas the structure of the component composition is taken from the particular component model. On one hand, TBP provides means for modelling internal behavior of individual components using threads and reactions to events on communication interfaces. On the other hand, restrictions can be imposed on the interfaces in the form of allowed sequences of method calls, such restrictions then form the properties of the model. These restrictions are called provisions.

In TBP, a notion of correctness is well defined for *closed* models, which are the models representing whole isolated application with a limited number of threads. The

correctness consists of two parts, a model is correct if it is free of *bad activity* error and *no activity* error. Put simply, the bad activity error may be described as an explicit violation of a provision, i.e. a method has been called in a way that is not allowed by a provision. Conversely, the no activity error corresponds to a situation where the program computation ends and where some events that have not been issued are expected. Put simply, no provisions have been explicitly violated, but at least one has not been fulfilled.

Additionally, the notion of refinement among components is also defined, it states that one component specification refines another if the former behaves correctly in all environments where the latter behaves correctly. The refinement is parametrized with the maximal number of threads that may be involved in the environments. The knowledge that a component refines another provides an opportunity to safely replace the latter with the former one in a component application. Nevertheless, such substitution is guaranteed to be safe only if the number of the threads in the application is less or equal to the parameter of the refinement.

Goals

This thesis focuses on a situation where no particular limit on the number of threads is known. Such a situation occurs when the provision of a component allows fully re-entrant [1] calls of some methods, i.e. methods may be called in parallel, and it is desirable to assure that the component or some composition of components will behave correctly in any environment which respects its provisions. In order to verify a model in this scenario, it is necessary to create an artificial component that will represent all the possible environments. Put simply, such a component must be able to perform all the possible combinations of events allowed by the provisions. As some provisions contain full reentrancy, it is obvious that the component will contain an unlimited number of threads.

Without specialized techniques, it is not possible to perform model checking on models with unlimited numbers of threads. The reason is that the state space is inherently infinite and thus it is not possible to simply traverse all the states. However, it is possible to create the artificial components so that they use an unlimited number of replicas of a finite number of particular threads (thread types). Then the infinite state space contains vast amount of symmetries induced by the thread replication.

The main goal of this thesis is to reduce these symmetries and propose a model checking technique for TBP models with unlimited numbers of threads. This work builds on results achieved in the field of model checking of concurrent boolean pro-

[1] It is worth mentioning, that full reentrancy can be expressed within the means for describing provisions in TBP – it explicitly contains full reentrancy operator.

grams, especially on counter abstraction technique [BMWK09, BHK+10]. This technique reduces state space symmetries by reducing the model state reachability problem to *coverability problem* of *vector addition systems with states* (VASS). However, it is not trivial to encode the TBP provisions as VASS in general as the provisions may also be inherently infinite, especially in the case of full reentrancy. This work focuses on a particular class of provisions with full reentrancy and its goal is to provide at least safe checking of provision-induced properties with respect to bad activity error.

It is also worth pointing out, that the described scenario is hypothetically not the only case where an unlimited number of threads may occur. Considering the most general case of a component model, it can for example allow several instances of a client component to communicate with a server component. Such a scenario might be modelled using dynamic reconfiguration of the component architecture that would allow adding new components. If the number of instances of the client component is not known in advance, the number of threads involved in this communication is potentially unlimited. The instances of the client component will be typically one-to-one copies. Thus such component composition also induces an infinite number of replicas of a finite number of threads and thus this work would also be applicable onto this scenario. However, such a construct is not supported in TBP and thus this work does not focus on it.

Contribution. In summary, this thesis accomplishes the following:

- A specific scenario of TBP model analysis with unbounded number of threads is focused on: given TBP specifications of components that are supposed to be used as a part of a closely unspecified application, it is desirable to verify whether these components will operate correctly with respect to bad activity under any environment which accomplishes provisions of the components.

- A safe model checking technique addressing such a scenario is proposed. This technique tackles inherent state space symmetries induced by unbounded replication of threads. The technique is safe in the sense that it discovers all potential bad activity errors. However, it can produce spurious errors, i.e. errors that do not relate to any actual error in the model.

- Along with the thesis, a proof-of-concept prototype is implemented. It covers all the key points of the proposed model analysis.

Structure of the text

In *Chapter 1*, the counter abstraction technique and its symbolic variant are briefly described. *Chapter 2* covers syntax and semantics of the TBP specification language,

defines TBP models and reviews analyses of the models according to error detection and refinement. In the next chapter, *Chapter 3*, the particular scenario of TBP analysis which this thesis focuses on is specified and captured in a specific TBP model called *incomplete TBP model*. Subsequently the model checking technique addressing the defined scenario is described and discussed in detail. *Chapter 4* reviews the proof-of-concept prototype implementation. Finally, *Conclusion* summarizes this work and the key the achievements.

Notation

This section reviews the common mathematical symbols and notations used in this work.

The sets of all natural numbers and integers are denoted using $\mathbb{N}$ and $\mathbb{Z}$ respectively, $\mathbb{N}_0$ denotes the set all of non-negative integers. The power set which is a set of all subsets of a given set S is denoted as $\wp(S)$.

Moreover, a special symbol ω, used in [KM69] to denote an unlimited counter value, also appears in the text. For ω holds:

- $\forall p \in \mathbb{Z} : p < \omega$
- $\forall p \in \mathbb{Z} : p + \omega = \omega$
- $\forall i \in \mathbb{N} : i\dot{\omega} = \omega$

In order to visually distinguish operations on numbers and vector, special symbols for operations are used. Let $v \in \mathbb{Z}^k$ be a k-dimensional vector of integers. If i is a natural number such that $1 \leq i \leq k$, then $v_{(i)}$ denotes i-th coordinate of v. The symbol $\leqslant$ denotes a point-wise comparison of vectors. Let $u, v \in \mathbb{Z}^k$ be vectors, then $u \leqslant v$ if and only if $\forall i \in \mathbb{N}, 1 \leq i \leq k \ : \ u_{(i)} \leq v_{(i)}$. Analogously, $\dot{+}$ a denotes point-wise addition of vectors. Let additionally $w \in \mathbb{Z}^k$, then $w = u \dot{+} v$ if and only if $\forall i \in \mathbb{N}, 1 \leq i \leq k \ : \ w_{(i)} = u_{(i)} + v(i)$.

Finite state machines are also often used in this thesis. Recall that a deterministic finite state machine is a tuple $(Q, \Sigma, \delta \subseteq Q \times \Sigma \times Q, s_0, F)$ where Q denotes a set of states, Σ denotes finite alphabet, δ is a transition relation, s_0 is the initial state and F is the set of final states. Often simply *finite state machine*, *finite automaton* or just *automaton* will be used to refer to the deterministic finite state machine. On the other hand, if a non-deterministic finite state machine occurs, its non-determinism will be properly emphasized. For completeness, recall that the non-deterministic finite state machine differs from the deterministic in the transition relation and initial states. A non-deterministic automaton can contain more than one initial state and its transition relation $\delta \subseteq Q \times \Sigma \times \wp(Q)$ can contain several transition starting at one state using the

same symbol from the alphabet. Let $\mathcal{A}$ be a finite state machine, then its set of states will be referred to using $Q_{\mathcal{A}}$, $\delta_{\mathcal{A}}$ will denote transition relation of $\mathcal{A}$ and $F_{\mathcal{A}}$ will be used to denote the set of final states of $\mathcal{A}$.

1. Symbolic Counter Abstraction

As a means allowing checking of models with unlimited number of threads, a technique called *symbolic counter abstraction* is utilized within this work. It was introduced in [BMWK09] as a symbolic application of *counter abstraction* symmetry reduction technique [ET99] for checking boolean programs, however, its key ideas are applicable to TBP model checking as well. Furthermore, in [Zha09], the approach was extended with a support of dynamic thread creation and unbounded numbers of threads.

1.1 Vector Addition Systems with States

The base of the symbolic counter abstraction consists in usage Vector Addition Systems with States (VASS) for solving thread state reachability problem. VASS is a variant of Vector Addition System (VAS) introduced in [HP76]. In the same work it was shown that VASS can be simulated by VAS.

Definition 1.1 (Vector Addition System with States)**.** A k-dimensional *Vector Addition System with States* (VASS) is a tuple $(\mathcal{S}, c_0, \delta)$ where $\mathcal{S}$ is a set of states, $c_0 \in \mathcal{S} \times (\mathbb{N}_0)^k$ is an initial configuration and $\delta \subseteq \mathcal{S} \times \mathcal{S} \times \mathbb{Z}^k$ is a transition relation.

Given a VASS (S, c_0, δ) and a pair of configurations $a = (s, u)$, $b = (t, v)$, there is a *transition* $a \to b$ if exists a vector $w \in \mathbb{Z}^k$ such that $(s, t, w) \in \delta$ and $u \dot{+} w = v \in (\mathbb{N}_0)^k$. Additionally, a sequence of configurations $c_1, c_2, ..., c_n$ is called a *path* from c_1 to c_n if for all the $i \in \mathbb{N}$, $i < n$ there is a transition from c_i to c_{i+1}. A configuration b is *reachable* from a configuration a if there is a path from a to b. Let (S, c_0, δ) be a VASS, then there is a natural question whether a configuration c is reachable in the VASS, i.e. whether c is reachable from c_0. A slight modification of this question forms the *VASS coverability problem*.

VASS coverability problem asks, for a given configuration $c = (s, u)$, whether a configuration $c' = (s, u')$ s.t. $u \leqslant u'$ is reachable.

The VASS coverability problem is proven decidable. In [KM69], Karp and Miller present an algorithm that builds a rooted labelled tree called *Karp-Miller tree* that represents a set of covered configurations of a VAS. In the same work, they show that the tree contains finite number of nodes (in other words, they guarantee termination) for any VAS. As any VASS can be simulated by using a VAS, the tree can be constructed for VASS and is finite as well.

1.2 Karp-Miller Tree

The following definition presents the tree extended to fit the VASS naturally.

Definition 1.2 (Karp-Miller Tree [KM69])**.** Let $\mathcal{W} = (\mathcal{S}, c_0, \delta)$ be a k-dimensional VASS and T be a rooted tree with labelled vertices; each vertex is labelled with an element of $\mathcal{S} \times (\mathbb{N}_0 \cup \{\omega\})^k$.

We say that T is *Karp-Miller Tree* for $\mathcal{W}$, $T = \mathcal{T}(\mathcal{W})$ if and only if

1. the root is labelled with c_0
2. let η be a vertex and (s_η, v_η) its label
   (a) if there is a vertex ν with label (s_ν, v_ν) such that $s_\nu = s_\eta$, $v_\nu \leqslant v_\eta$ and ν is on the path from the root to η, then η is a leaf (i.e. it has no successors);
   (b) otherwise successors of η are in one to one correspondence with $(s_\eta, t, u) \in \delta$ such that $v_\eta \dot{+} u \in (\mathbb{N}_0 \cup \{\omega\})^k$. The successor ν corresponding to (s_η, t, u) is labelled with (t, w). For every $i \in \{1, 2, ..., k\}$, $w_{(i)}$ is set to
      - ω if there exists vertex ϕ labelled with (s_ϕ, y) on the path from root to ν such that $y \leqslant (v_\eta \dot{+} u)$ and $y_{(i)} < (v_\eta \dot{+} u)_{(i)}$;
      - $(v_\eta \dot{+} u)_{(i)}$ otherwise.

Obviously, a configuration $c = (s, u)$ is covered in a VASS $\mathcal{W} = (\mathcal{S}, c_0, \delta)$ exactly if there exists a label (s, v) on a node in the Karp-Miller tree for $\mathcal{W}$ such that $u \leqslant v$.

For example, consider a 3-dimensional VASS $\mathcal{W} = (\{p, q\}, (p, (1, 0, 0), \delta)$ where

$$\delta = \{ \quad (p, q, (-1, 1, 0)), \quad (p, q, (-1, 0, 1)) \\ (q, p, (2, 0, -1)), \quad (q, p, (2, -1, 0)) \quad \}$$

Figure 1.1 depicts this VASS and Figure 1.2 shows the Karp-Miller tree constructed according to Definition 1.2.

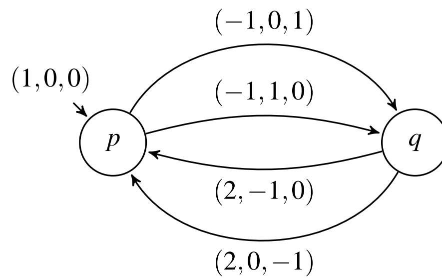

Figure 1.1: Example of VASS

1.3 Counter Abstraction

The counter abstraction tackles state space symmetries induced by thread replication by reducing the thread-state reachability problem to the VASS coverability problem. The key idea is that shared states (i.e. valuation of shared variables) of a program

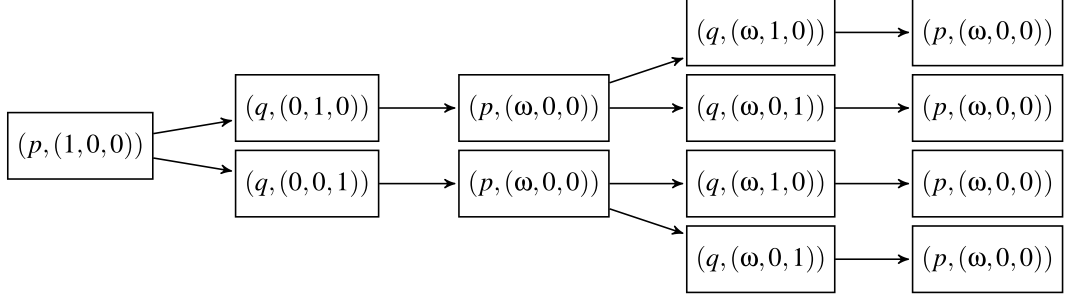

Figure 1.2: Example of Karp-Miller tree

are represented with states of VASS, whereas local states (i.e. valuation of thread local variables and program counter) are enumerated and coupled with elements of the vector. A configuration (s,v) then represents the state of the program execution such that the values of shared variables are captured in s and for each i there is $v_{(i)}$ threads residing in the i-th state. Hence the name since the vector coordinates act as the counters of threads[1]. Single step of the program, which in particular means that a thread performs an instruction and thus it changes its local state and possibly assigns new value to a shared variable, is represented by $(s,t,w = (0,0,...,0,-1,0,...,0,1,0,...)) \in \delta$. The shared state is changed from s to v and the thread state shift is captured with -1 (leaves the state) and 1 (enters another state).

Having such VASS model of a program, the coverability of a configuration (s,u) answers the question whether the point of program execution where the values of the shared variables correspond to s and in the i-thread state reside at least $u_{(i)}$ threads $(\forall i, 1 \leq i \leq k)$ is reachable. Hence the thread-state reachability problem for the program can be reduced to the VASS coverability problem.

The explicit approach to counter abstraction constructs thread transition diagrams and then creates a VASS explicitly. Obviously, the dimension of the VASS (and in turn the number of counters) corresponds to the number of all the possible valuations of local variables. As the number of valuations grow exponentially with the number of variables, the number of counters may be enormously high. This problem was identified in [BMWK09] as *local state space explosion*.

The symbolic variant addresses the local state space explosion with interleaving the model checking phase (Karp-Miller tree construction) with the state space exploration phase, in other words, it creates the VASS transitions *on-the-fly*. Moreover, it does not retain whole vectors, but only non-zero counters – a zero-valued counter means that no thread resides in the appropriate state.

The step towards dynamic thread creation and unbounded number of threads leads through defining some points where the threads can arise. Specifically, such point is

[1] In some literature counter abstraction denotes limiting possible counter values or abstraction of the counter domain, however, we keep terminology of [BMWK09].

represented by $(s,t,w) \in \delta$ where $\sum w_{(i)} > 0$. Since the Karp-Miller tree is finite for every VASS, the thread state reachability can be reduced to the VASS coverability even for this case.

2. Threaded Behavior Protocols

As it was already mentioned in the introduction, TBP is a language for modelling behavior of components. It aims to be as close as possible to the mainstream imperative programming languages in syntax and semantics as well. TBP concentrates on providing means to specify behavior of individual components, whereas the architecture of component composition is supposed to be provided by an underlaid component model. This chapter reviews TBP as it was proposed in [KPS09] and elaborated in detail in [Poc10].

2.1 Syntax

TBP specification of a component consists of five parts: type declarations, variable declarations, provisions, reactions and threads. The reactions and threads describe the internal behavior of the component in the imperative manner, whereas provisions specify behavior of the environment assumed by the component. The assumptions are expressed as allowed sequences of calls of methods provided by the component. It is worth to mention, that TBP is supposed to be used as an extension of a component model. The information about provided and required methods (or interfaces which are in fact groups of methods) is supposed to be obtained from the underlaid component model. From the point of view of the specification, it is enough to assume that there are three sets Σ_{prov}, Σ_{req} and Σ_{int} where the first contains names of provided methods, the second contains names of required methods and the last contains names of internal methods (methods that are neither provided nor required).

```
component ComponentName {
        types {
                ...
        }
        vars {
                ...
        }
        provisions {
                ...
        }
        reactions {
                ...
        }
        threads {
                ...
        }
}
```

2.1.1 Type Declarations

The *types* section defines enumeration types for variable declarations. An enumeration type declaration consist of a name and a list of enumeration values. The following fragment declares two enumeration types **`result`** and **`mode`**. The former consist of enumeration values **`OK`** and **`FAILED`** and the later contains three enumeration values **`READY`**, **`RUNNING`** and **`STOPPED`**.

```
types {
        result = { OK, FAILED }
        mode = { READY, RUNNING, STOPPED };
}
```

2.1.2 State Variables

State variables are declared in the *vars* section. These variables are accessible only to the component, however, they are shared among threads and reactions of the component. The state variables will be often referred to as *global* variables. A state variable declaration consists of a type, a name and an initial value. The following fragment declares a variable named **`currentMode`** of the type **`mode`** with the initial value **`READY`**.

```
vars {
        mode currentMode = READY;
}
```

Additionally, there is a special type of variable – *mutex*. A mutex variable serves as synchronisation object which allows threads to achieve mutual exclusion.

2.1.3 Reactions

The *reactions* section contains description of the behavior of the component performed in reaction on a method call. Each reaction specification consists of the method name, a declaration of arguments and a reaction body. Optionally, the arguments may be followed by a return type. The body begins with declaration of local variables followed by specification of the behavior. A local variable declaration specifies a type, a name and an initial value of the variable. The local variables are accessible only to the body of the reaction. Moreover, each execution has its own copies of the local variables, thus the values are not shared among them. Technically speaking, the arguments are just a special case of local variables with the difference, that the actual values are assigned during the method call.

```
reactions {
        interfaceName.methodName(
                        argType1 argName1,
                        argType2 argName2,
                        ...):returnType {
                localVarType1 localVarName = INIT_VALUE;
                ...
        }
}
```

The specification of the reaction behavior consists of elementary actions composed together using control flow operators.

Elementary Actions

- Method call – `i.m(v1, v2, ...)`
  The method `m` on the interface `i` is called with parameters $v1$, $v2$, etc. The execution is switched to the reaction of the method and the current execution is resumed as the reaction of the called method finishes.

- Return – `return value`
  Terminates the reaction and the return value may be assigned to a variable at the call site.

- Assignment – `var = val`
  The current value of the `var` variable is changed to the current value of `val`, which may refer to a constant, a variable or a method call. In the case of a method call, the current value is the return value.

Control Flow Operators

- Sequence operator

  ```
  a;b
  ```

  The operands (`a`, `b`) are executed sequentially.

- Alternative operator

  ```
  if (condition) {
          thenBranch
  } else {
          elseBranch
  }
  ```

Just one of the operands (**thenBranch**, **elseBranch**) is executed depending on the result of **condition**. The condition is an expression defined using following rules:

- **?** is a condition expression. It represents non-deterministic choice, the result is either *true* or *false*.
- **var == val** is a condition expression if **var** is either a local variable or a state variable and **val** is either a local variable, a state variable or a enumeration constant. The result is *true* if and only if the current values of **var** and **val** are equal.
- **!C** is a condition expression if **C** is a condition expression. The result is *true* if and only if the the result of **C** is *false*.
- **C1&&C2**, **C1||C2** are condition expressions if both **C1** and **C2** are condition expressions. The result of the former is *true* if and only if the results of both **C1** and **C2** are *true*. The result of the later is *true* if and only if the result of **C1** is *true* or the result of **C2** is *true*.

• Switch operator

```
switch (var) {
        case const1: branch1
        case const2: branch2
        ...
        default: defaultBranch
}
```

Exactly one of the operands (**branch1**, **branch2**, ..., **defaultBranch**) is executed depending on the current value of the **var** variable, which may be a local variable or a state variable. In particular, the *i*-th branch is executed if the current value of **var** is equal to the corresponding the enumeration constant **const***i*. If the current value of **var** is not equal to any constant in the list, the **defaultBranch** operand is executed.

• Repetition operator

```
while (condition) {
        block
}
```

The operand (**block**) is executed repetitively as long as the result of the condition is *true*. The condition is defined in the same way as in the case of the alternative operator.

- Synchronization operator

```
sync (mutex) {
        block
}
```

The operand (**`block`**) is executed in a synchronized way with respect to the state variable **`m`** of the special built-in type **`Mutex`**. In particular, only one block synchronized on a single mutex variable may be executed at a moment.

2.1.4 Threads

The *threads* sections specifies a list of threads in the component. Each thread is defined with a name and a body. The body begins with a specification of local variables which is followed by a specification of the thread's behavior. The behavior is described in the same way as in the case of reactions. The only difference is that the return statements are not allowed.

The threads represent a source of activity of the component. Any component without threads is only capable to perform reactions on method calls from the environment. Conversely, threads allow components to perform actions on their own. A significant feature is that threads are executed in parallel.

2.1.5 Provisions

The *provisions* section declares the assumptions posed on the environment. Each assumption is defined as a set of allowed sequences of calls and returns of the component's provided methods. The component is supposed to be used in such environments that do not violate the assumptions.

A single provision declaration consists of an expression defining the set of allowed sequences and a list of methods that are covered by the provision. The following fragment declares two provisions, the former covers the methods **`m`** and **`n`** of the interface **`i`** and the latter covers the method **`m`** of the interface **`j`**.

```
provisions {
        { expr1 } for { i.m, i.n }
        { expr2 } for { j.m }
}
```

Provision Expressions

Each provision expression consists of basic building blocks – elementary expressions – composed together using provision operators. An elementary expression represents

two consequent events – a method call and the corresponding return. The specification of an elementary expression consists of a method name, parameters specification and a return value where appropriate. The specification of parameters consists of a list of enumeration constants (of respective types to the arguments of the method) and the return value is specified by a single enumeration constant. Both the parameters and the return value are optional.

```
provisions {
        { calc.sum(POS, ZERO): POS } for { calc.sum }
        { calc.sum() } for { calc.sum }
}
```

The above fragment contains two provision expressions. The first defines single allowed sequence which consists of a call of the `sum` method of the `calc` interface with two parameters `POSITIVE` and `ZERO` followed by the corresponding return with value `POSITIVE`. The second defines the set of all the sequences of two events such that the first event is a call of the method with an arbitrary combination of the possible values of the parameters and the second event is the corresponding return with an arbitrary value (of the enumeration type associated with the reaction). Note that if the `sum` method declared empty list of arguments and no return value, the expression would define a single sequence of the non-parametrized call followed by the corresponding return (without a value).

Operators. The provision operators allow creating more complex expressions from the elementary expressions. There are three operators commonly used in regular expressions – sequence, alternative and repetition. Additionally, parallel and reentrancy operators are allowed.

- Sequence operator – `;`
  The sequence operator, which is a binary operator, defines the set of allowed sequences as the set of concatenations of the allowed sequences of the operands.

- Alternative operator – `+`
  This operator defines the set of allowed sequences as the union of the allowed sequences of the operands. It is also a binary operator.

- Repetition operator – `*`
  The repetition operator is an unary operator. Any (even null) repetition of the sequences in the operand's set of allowed sequences is allowed in the resulting expression. For any provision expression `P`, the expression `P*` is equivalent to λ `+ P + (P;P) + (P;P;P) + ...` where λ stands for an empty sequence of events.

- And-parallel operator – `|`
  The and-parallel operator, which is also a binary operator, allows all the possible interleavings of the sequences allowed by the operands. For illustration, let `P` and `Q` be provision expressions each allowing single sequence of events `a;b` and `c;d` respectively. Then the provision expression `P | Q` allows six sequences of events: `(a;b;c;d)`, `(a;c;b;d)`, `(a;c;d;b)`, `(c;a;b;d)`, `(c;a;d;b)` and `(c;d;a;b)`.

- Or-parallel operator – `||`
  `P || Q` is equivalent to `(P | Q) + P + Q`.

- Limited reentrancy operator – `|`n for $n \in \mathbb{N}$
  `P|`n stands for `P || P || ... || P` where `P` occurs n times.

- Full reentrancy operator – `|*`
  `P|`n stands for `P || P || ...` . In this case, any sequence formed as an interleaving of an arbitrary number of the sequences allowed by the operand is allowed by the expression.

2.2 Semantics

The syntactical framework described in the previous section together with information from an underlaid component model form a TBP specification. The semantics of such a specification is then represented by TBP model, which is a mathematical structure capturing the specification. The TBP model is derived from the specification for each component separately and these partial models are composed together using information from the underlying component model.

The composed model is then a subject of analysis – either error detection or refinement analysis depending on whether the model is closed or open respectively. The closed model does neither issue nor expect any externally observable activity, in other words it represents a closed standalone application. Conversely, the open model corresponds to a composition of components or a single component which is intended to interact with other components – an environment. The open model can represent a part of an overall application or it can be imagined as a library used within the application.

2.2.1 TBP Model

The TBP model is a mathematical structure that precisely represents the semantics of the TBP specification. It fills the gap between rich syntax and precise semantics and captures all the aspects of the specification.

In following definitions E denotes a set of enumeration types and V a set of variables. A variable v is an entity with associated type $e \in E$, domain Dom_v which is equal to the set of constants Dom_e declared in type e, and an initial value $Init_v \in Dom_v$. Additionally Dom_E will denote a set of all constants

$$Dom_E = \bigcup_{v \in V} Dom_v$$

Note that a mutex m is a special case of variable with $Dom_m = \{\texttt{UNLOCKED}, \texttt{LOCKED}\}$ and $Init_m = \texttt{UNLOCKED}$. Any variable takes values from its domain Dom_v which is captured by valuation functions.

Definition 2.1 (Valuation [Poc10]). Let V be a set of variables then $\gamma_V : (V \cup Dom_E) \rightarrow Dom_E$ is a *valuation function* over V if and only if

$$(\ \forall a \in V : \gamma_V(a) \in Dom_a\ ) \ \wedge\ (\ \forall e \in Dom_E : \gamma_V(e) = e\ )$$

In addition, if

$$\forall w \in W : \gamma_V(w) = Init_w$$

then γ_V is an *initial valuation* which is denoted γ_V^0.

Modified valuation $\gamma_V[v \leftarrow e]$ where $v \in V$ and $e \in Dom_v$ is a valuation function such that

$$\gamma_V[v \leftarrow e](a) = \begin{cases} e & \text{if } a = v \\ \gamma_V(a) & \text{otherwise} \end{cases}$$

Definition 2.2 (Guards [KPS09, Poc10]). Let V be a set of variables. A *guard* over V is a finite expression derived using following rules:

- $true$ is a guard;
- $v{=}{=}l$, where $v \in V$ and $l \in Dom_v$ is a guard;
- if X and Y are guards, $X \wedge Y$, $X \vee Y$ and $\neg X$ are also guards.

Actual value of the guard g under valuation γ_V is denoted as $\gamma_V(g)$ and the set of all guards over V is denoted as G_V.

The guards in Definition 2.2 are simple logical expressions over the variables. The actual value of a guard takes boolean values $true$ or $false$ and it is defined using following rules

- $\gamma_V(true) = true$
- $\gamma_V(v{=}{=}l) = true \iff \gamma_V(v) = \gamma_V(l)$

- $\gamma_V(\neg X) = true \iff \gamma_V(X) \neq true$
- $\gamma_V(X \vee Y) = true \iff \gamma_V(X) = true \vee \gamma_V(Y) = true$
- $\gamma_V(X \wedge Y) = true \iff \gamma_V(X) = true \wedge \gamma_V(Y) = true$

assuming that $v{=}{=}l$, X and Y are guards over V.

Definition 2.3 (Assignment [Poc10])**.** Let V be a set of variables. *Assignment* over V is a label in the form

- $v{\leftarrow}c$ where $v \in V$ and $c \in Dom_v$
  assigning constant c of corresponding type to v
- $v{\leftarrow}w$ where $v, w \in V$ and $Dom_v = Dom_w$ assigning actual value of variable w to v

Let A_V be a set of all assignments over V.

The assignments are supposed to change the values of the variables. Let $v{\leftarrow}w$ be an assignment over V and let γ_V be a valuation of variables before the assignment. After the assignment is performed, the valuation will be $\gamma_V[v \leftarrow \gamma_V(w)]$.

Definition 2.4 (Labelled Transition System with Assignments [KPS09, Poc10])**.** A *Labelled Transition System with Assignments* is a six-tuple $(S, s_0, F, \delta, \Sigma, V)$, where S is a set of states, $s_0 \in S$ is an initial state, $F \subseteq S$ is a set of final states, Σ a set labels, V a set of variables and $\delta \subseteq S \times G_V \times \Sigma \times \wp(A_V) \times S$ is a transition relation.

Note that Definition 2.4 slightly differs from the one presented in [KPS09, Poc10] in the point, that it allows more than one assignment to be attached to a transition. In fact, it allows a transition to contain a non-empty label and a non-empty set of assignments at once. Covnersely, the original LTSA definition restricted the transitions so that they could contain just one assignment or a label ($\delta \subseteq S \times G_V \times (\Sigma \cup A_V) \times S$). Clearly any original LTSA can be expressed as an LTSA along Definition 2.4.

On the other hand, the proposed extension of the LTSA structure can bring some problems: consider a transition that contains two different assignments to one variable. Then the value of the variable after the assignments depends on the order in which were the assignments performed. Therefore the following notion is introduced. Let $l = (S, s_0, F, \delta, \Sigma, V)$ be a LTSA, then l is *independent of the order of assignments* if and only if for all transitions $(s, g, l, a, s') \in \delta$ such that $a \neq \emptyset$ the following holds

$$\forall (u \leftarrow v), (x \leftarrow y) \in a : (u = x \Rightarrow v = y) \wedge (v = x \Rightarrow u = v) \wedge (u = y \Rightarrow x = y)$$

If the assignments in the form $x \leftarrow x$, which actually have no effect, are not considered, the formula in fact says that no variable occurs more than once on the left side among

all the assignments on any transition. Moreover, if a variable appears on the right side of an assignment on a transition, the variable does not appear on the left side of any assignment on the same transition. Thus the order of assignments does not matter.

Moreover, let $\tau \in \Sigma$ denote an empty label, then l is *strict* if and only if for all transitions $(s, g, l, a, s') \in \delta$ the size of a is at most 1 and if $g \neq \tau$ then $a = \emptyset$. The strict LTSAs corresponds to the original LTSAs definition. In the following text, only the LTSAs independent of the order of assignments will be considered.

Definition 2.5 (LTSA computation state [Poc10])**.** Let $l = (S, s_0, F, \delta, \Sigma, V)$ be a LTSA. A tuple (s, γ_V) is a *computation state* of l if $s \in S$ and γ_V is a valuation over V. The tuple (s_0, γ_V^0) is denoted as *initial computation state*.

Definition 2.6 (Enabled LTSA transition [Poc10])**.** Let $l = (S, s_0, F, \delta, \Sigma, V)$ be a LTSA and $c = (s, \gamma_V)$ be a computation state of l. Then a transition $(s, g, l, a, s') \in \delta$ is called *enabled* in the computation state c if $\gamma_v(g)$ is $true$.

The LTSA is a crucial structure for representing control flow of imperative parts of TBP specification – threads and reactions. In the context of TBP models, the labels will represent issuing method calls. Let Σ contain method names, $\Sigma^{\uparrow}$ contains events for issuing a method call, $\Sigma^{\downarrow}$ contains events for accepting a result from a method call and $\Sigma^{\uparrow/\downarrow} = \Sigma^{\uparrow} \cup \Sigma^{\downarrow}$.

Let $m \in \Sigma$ be a method name, then $\uparrow m$ is an event for issuing a call of the method m and $\downarrow m$ is an event for accepting a result from the method m.

Definition 2.7 (Parametrized label [Poc10])**.** Let Σ be a set of labels and P a set of parameters. Then the set of parametrized labels is defined as
$\Sigma_P = \{(m, \langle v_1, v_2, ..., v_n \rangle) \mid m \in \Sigma, n \in \mathbb{N}_0, v_i \in P\}$.

Then $\Sigma^{\uparrow}_{V \cup Dom_E}$ denotes a set of call events parametrized by all possible combinations of variables and constants from all the enumeration types.

For the purpose of TBP Model definition, let $LTSA^{\Sigma}_{V,E}$ denote the set of all the LTSAs using variables V and labels $\Sigma^{\uparrow}_{V \cup Dom_E} \cup \{\tau\}$ where τ denotes an empty label. Thus any non-empty label on a transition in such LTSAs represents issuing a method call including parameters. Since not only the strict LTSAs are considered, it is worth mentioning, that the assignments are performed before the method call is issued, as the parameter values may be affected by the assignments. However, in the TBP models derived directly from TBP specifications only strict LTSAs will occur. The return value of the method call is retained in a special purpose variable Ret. In order to assign the return value of the method call to a regular user-defined variable v, the subsequent transition is supposed to contain assignment $v \leftarrow Ret$.

Definition 2.8 (TBP Model [KPS09, Poc10])**.** A *TBP Model* is a five-tuple (Σ, P, R, T, G), where:

- $\Sigma = (\Sigma_{prov}, \Sigma_{req}, \Sigma_{int})$ denotes disjunct sets of provided, required and internal method names used in the model

- G is a set of state variables

- P is a set of provisions $\{P_1, P_2, P_3, ..., P_n\}$ taking the form $P_i = (filter^{P_i}, traces^{P_i})$ where $filter^{P_i} \subseteq \Sigma_{prov}$ specifies methods observed by the provision and $traces^{P_i}$ specifies the allowed finite sequences of events in $\left(filter^{P_i}\right)^{\uparrow/\downarrow}_{Dom_E}$.

- R is a partial function: $(\Sigma_{prov} \cup \Sigma_{int}) \rightarrow (L, \mathbb{N} \rightarrow L, LTSA^{\Sigma_{int} \cup \Sigma req}_{G \cup L,E})$ representing mapping of method names to their local variables (L), a parameter mapping function and a reaction in the from of LTSA.

- T is a set of threads $T_1, T_2, T_3, ..., T_m$, where $T_i \in (L, LTSA^{\Sigma_{int} \cup \Sigma_{req}}_{G \cup L,E})$ is a tuple specifying a set of local variables and the behaviour of the i-th thread in the form of LTSA.

Note that a set of allowed finite sequences $traces^{P_i}$ can be potentially infinite and as the provision in general can contain full reentrancy operator, it is not possible to represent it using finite state machines. Rather, let it be represented by the provision expression itself which can be converted to the particular structure on demand. Thus obtaining provisions from a specification is straightforward in general, the only difference between provision expressions stated in the specification and in the model is that every elementary expression is replaced with a the corresponding couple of events (a method call and the corresponding return).

Deriving the variables and method names is also straightforward as they are directly stated in the specification. The only part which deserves a deeper description is the construction of the LTSAs for threads and reactions. The following describes the conversion in a bottom-top manner.

The elementary actions (assignment, method call, return) represent single LTSA transitions. The only exception is an assignment of a return value of a method call to a variable. It is represented by a sequence of two transitions – the first issues the method call and the second assigns value of the special *Ret* variable to the target variable. Note that transitions that contain an assignment will be equipped with τ label. Additionally, the assignment set on transition corresponding to `NULL` action is empty and the label is τ.

The control flow operators are processed in a similar way to the way finite state machines are constructed from regular expressions. The sequence operator (`;`) corresponds to concatenation of the LTSAs of operands.

The `if-then-else` and `switch` operators correspond to the alternative operator in regular expressions. The entry transitions to the individual branches will be equipped

by corresponding guards in case of deterministic conditions (not `?`). Note that the guard on the `else` branch is a negation of the `if` condition. In the case of `switch` on a variable v, the branch corresponding to a case with constant c will be guarded by $v{=}{=}c$ and the `default` case will be guarded by the negation of conjunction of guards corresponding to all the `cases`.

The `while` operator is similar to the Kleene star in regular expressions (`*`) with the difference that there will be an *exit* transition starting at the initial state of the LTSA for the repeated block and pointing to a new final state. The exit transition will contain no assignments and will be labelled with τ. Any transition to the original final states will be redirected to the starting state if it is not labelled with a return label. In the case of a deterministic condition by the `while` statement, the transitions from the initial state will be labeled with a guard corresponding to the condition (and its negation in case of the exit transition).

The `sync` operator on a mutex `m` adds a new initial state and a new final state to the LTSA of the synchronized block. It connects the new initial state with the original one by using a transition guarded with `m` $==$ `UNLOCKED` and with single assignment `m` $\leftarrow$ `LOCKED`. The original final states (which will no longer be final states) are connected with the new one by using transitions with the assignment `m` $\leftarrow$ `UNLOCKED`. For the sake of simplicity, this construction assumes that no `return` statements appear in the synchronized block – in this simple way of construction the `return` statement could cause deadlock as the mutex would not be properly unlocked. However, it is possible to propose a modified construction which will support `return` within `synchronized`, but it would be a bit more complex.

Note that the transitions derived from the elementary actions contain just one assignment, a call label or a return label. As all the control flow operators introduce at most one assignment and no label per a new transition, the derived LTSA is strict.

Composition

So far, the way of deriving the TBP model from a TBP specification of a single component has been described. Consequently, as the components are composed together within, a definition of composition over the TBP models is presented in the following.

First, note that two TBP models can not provide the same methods in order to keep their composition well defined. The duplicate provided methods could be subsequently required by a single component and such a *broadcast* semantic is not well defined. Similarly, the internal methods of the models should also be unique in order to keep the method binding consistent. Definition 2.9 states these requirements formally. Then Definition 2.10 provides the composition operation itself.

Definition 2.9 (Composable TBP Models [Poc10]).
Let $A = (\Sigma', P', R', T', G')$, $B = (\Sigma'', P''', R'', T'', G'')$ be TBP models. We say that A and B are *composable* iff $\Sigma'_{prov} \cap \Sigma''_{prov} = \emptyset$ and $\Sigma'_{int} \cap \Sigma''_{int} = \emptyset$.

Definition 2.10 (TBP Composition [Poc10]).
Let $A = (\Sigma', P', R', T', G')$, $B = (\Sigma'', P'', R'', T'', G'')$ be compsable TBP models. Then the *composition* is defined as

$$A \oplus B = ((\Sigma_{prov}, \Sigma_{req}, \Sigma_{int}), P' \cup P'', R' \cup R'', T' \cup T'', G' \cup G'')$$

where

$$\Sigma_{prov} = \Sigma'_{prov} \cup \Sigma''_{prov}$$
$$\Sigma_{req} = (\Sigma'_{req} \cup \Sigma''_{req}) \setminus (\Sigma'_{prov} \cup \Sigma''_{prov})$$
$$\Sigma_{int} = \Sigma'_{int} \cup \Sigma''_{int}$$

Note that the model after composition provides all the methods that were provided in the original methods. This allows a provided method to be required from more than one component.

2.2.2 Closed Model Analysis

A closed TBP model is specific in that it does not provide nor require any methods from the outer environment. In fact, it is fully isolated from outer world and any activity in the environment does not affect the behavior of the closed model. Moreover, the number of threads in the closed model is known in advance and is finite. Thus full reentrancy can be removed from provision expressions and they can be converted to finite state machines.

Let $M = (\Sigma, P, R, T, G)$ be a closed TBP model and the number of threads be k. Then any provision expression for $P_i \in P$ can be modified in the way, that every occurrence of $Q|*$ is replaced with $(Q;(Q*))\,|k$ for arbitrary Q. Let $P_{i\perp k}$ denote such a modified provision expression. The expression than does not contain full reentrancy and thus it can be rewritten as a regular expression and in turn converted into a deterministic finite state machine – let $FSM_{P_{i\perp k}}$ denote such a machine. Finally, let $M_{\perp k}$ denote the TBP model derived from M in the way that the set of allowed traces is for every provision represented by a deterministic finite state machine, i.e. $traces^{P_i} = FSM_{P_{i\perp k}}$.

In order to detect errors in a closed model, a closed computation is formally defined in [Poc10]. This work reviews it just informally. In the cited work the computation is inspired by stack-based execution model, where the stacks are used to capture method calls correctly. However, as mentioned in [KPS09], there is an alternative approach

which utilizes the fact, that there are no recursive calls, and thus reactions may be *inlined* to the corresponding call sites. The alternative approach will be used later in this work, within symbolic counter abstraction, where it largely simplifies the design.

Intuitively, the stack-based computation defines a computation state composed of stacks St and a valuation of global variables γ_G. St is a set of stacks which are in one to one correspondence to the threads, let S_i denote a stack corresponding to the thread T_i. Each stack is an ordered sequence of *local* LTSA computation statess. A local computation state is in fact a LTSA computation state with the difference that the valuation function is defined only over local variables of the corresponding thread or reaction. The top of the stack (lastly inserted element) captures the current execution scope of the thread. The subsequent element (if any) captures the scope which the current scope (a reaction) was invoked from.

There is a single initial computation state for each TBP model, which is given by γ_G^0 and a set of stacks such that each stack contains just one element representing the initial local LTSA computation state of the corresponding thread's LTSA. Individual steps of the threads then form transitions between the computation states. A computation transition is determined by a thread and a transition of the thread's LTSA which is enabled in current computation state. Let $T_i = (L_i, \Gamma_i)$ be such a thread and let t denote such an transition. Let S_i be the stack of T_i, $c = (s, \gamma_L)$ be the top-most element of S_i and γ_G be the valuation of global variables of the current computation state. Let t be enabled in $(s, \gamma_{G \cup L})$. Than in the subsequent computation state given by t the c stack element will be modified so that $c := (s', \gamma_L)$.

Additionally, if t contains some assignments a then the local valuation is modified so that $c := (s', \gamma_L[a])$ where $\gamma_L[a]$ denotes sequential modification of the valuation with all the elements of a which assign a value to a local variable in arbitrary order (note that it is well defined since all the LTSAs are independent of the order of assignments). The global valuation is modified analogously with assignments to the state variables.

Moreover, if t contains a call label, it pushes new element on the stack S_i, so that it is initial local LTSA computation state of the called reaction with single exception – the valuation of local variables corresponding to parameters are modified by values of variables under $\gamma_{G \cup L}$ which were used as parameters in the label. Conversely, if the transition contains return label with a parameter r then the local valuation in the lower stack element is modified so that value of Ret will be $\gamma_L(r)$. Finally, if s' is a final state, the top most element of the stack is removed.

A computation trace of the TBP model is then a sequence of computation states where each state is derived using a computation transition from the previous (except the initial one), the sequence starts with the initial computation state. A computation trace may end in several different ways. First, if it ends with a computation state that contains only empty stacks, the trace will be called *terminating*. If it ends with a state

that contain some non-empty stacks and all the top most elements of stacks correspond to LTSA states with no enabled outgoing transitions, the trace will be called *stuck*. Finally, it may not end at all – then it is *infinite*. Note that a terminating trace corresponds to a successful execution, a stuck trace represents occurrence of a deadlock and an infinite track signalizes, that there is a infinite activity in the model – e.g. an endless loop. The stuck traces are considered erroneous, whereas some infinite tracks may be desirable in some models – they will be distinguished later.

For the purpose of the communication error detection, it is necessary to track the call and return events executed along the computation. One can imagine that there is a running tape along a computation and each method call and return is recorded on the tape in time of its execution.The labels are recorded with current values of variables used as parameters of the labels. Let C be a computation trace, then $tape_C$ will denote the tape, a sequence of labels parametrized by Dom_E issued along the trace C. Note that if C is a infinite track and $tape_C$ is finite, then it corresponds to a situation, where the model stuck in an endless loop without any observable activity. Such case of infinite track is considered erroneous, whereas an infinite C with infinite $tape_C$ may be desirable in some reactive models.

In order to compare $tape_C$ to a provision P_i it is worth removing all the labels that are note observed by the provision and denote the rest $tape_C^{P_i}$ – it will contain just labels from $\left(filter^{P_i}\right)_{Dom_E}^{\uparrow/\downarrow}$. Then the tape can be compared to provision and the communication errors can be defined as follows.

Definition 2.11 (Bad Activity)**.** Let C be a computation trace of a TBP model $M = (\Sigma, P, R, T, G)$ and $Q = \left(filter^Q, traces^Q\right) \in P$. Then C is *free* of *bad activity error with respect to Q* if for an arbitrary finite prefix p of the $tape_C^Q$ there is $t \in traces^Q$ such that p is a prefix of t.

C contains a *bad activity error with respect to Q* if it is not free of bad activity error with respect to Q. Moreover, C contains *bad activity error* if it contains a bad activity error with respect to at least one $R \in P$. The TBP model M contains *bad activity error* if at least one of its computation traces contains bad activity error.

Definition 2.12 (No Activity)**.** Let C be a computation trace of a TBP model $M = (\Sigma, P, R, T, G)$ and $Q = \left(filter^Q, traces^Q\right) \in P$. Then C is *free* of *no activity error with respect to Q* if $tape_C^Q$ is infinite or $tape_C^Q \in traces^Q$.

C contains *no activity error with respect to Q* if it is not free of no activity error with respect to Q. Moreover C contains *no activity error* if it contains no activity error with respect to at least one $R \in P$. The TBP model M contains *no activity error* if at least one of its computation traces contains no activity error.

Summarizing all the error types, a TBP model M contains an error if

- M generates a stuck computation trace (deadlock)

- M generates an infinite trace C s.t. $tape_C$ is finite ("void" endless loop)
- M contains *bad activity error*
- M contains *no activity error*

2.2.3 Open Model Analysis

The analysis of an open TBP model presented in [Poc10] is based on a refinement relation over TBP models. A TBP model I is said that refines another TBP model S if I behaves correctly in any environment which S behaves correctly in. For such relation, there are means for deciding whether one model refines another provided in the referred work. These means are parametrized with a maximal number of threads that are allowed in any environment which is considered for the refinement.

The S model is supposed to be a relatively simple model that captures only significant points of the implementation I, which typically captures is a more complex component or a composition of components. The refinement relation allows vertical decomposition of an complex application in a hierarchical component system. In particular, an implementation I can be checked whether it refines a specification S. Then I may be replaced by S in a more complex fraction of the overall application which included I. Then the more complex part can be again checked for a refinement or, if it forms a closed model, checked for correctness.

In this section the refinement relation and the means for deciding whether two components are in the relation will be briefly reviewed. For more details a reader is kindly referred to the original work [Poc10].

The refinement relation parametrized with maximal number of threads in environment k is defined as follows. Let S and I be two TBP models and E_k be a set of all TBP models with maximally k threads s.t. $\forall e \in E_k$ the composition $e \oplus I$ form a closed model. Let's assume that S is such that $\forall e \in E$ also $e \oplus S$ forms a closed model. Then S refines I with respect to bad activity up to k threads if $e \oplus I$ does not contain bad activity error for each $e \in E_k$ such that $e \oplus S$ does not contain bad activity error. Similarly S refines I up to k threads with respect to no activity up to k threads if $e \oplus I$ does not contain no activity error nor any stuck computation trace for each $e \in E_k$ such that $e \oplus S$ does not contain no activity error nor any stuck computation trace. Then S refines I up to k threads if S refines I with respect to bad activity up to k threads and with respect to no activity up to k threads. Finally, S refines I if and only if S refines I up to k threads for all $k \in \mathbb{N}$.

The process of deciding whether two models are in the refinement relation is based on *provision-driven computation* and subsequent comparison of *observation projections* of those computations. The provision-driven computation is similar to the stack

based computation for closed model with the difference that it includes actions derived from the provisions of the model's provided methods.

The state of provision-driven computation in addition to the stack based closed computation state contains a state of an automaton derived from the provision expressions. All provisions of the model are combined together to form a deterministic finite state machine FSM_P which accepts exactly the words that are allowed by the provisions. As each provision forms a regular language (because the number of threads is limited) at is clear that such machine can be constructed. The transitions induced by the model itself correspond to transitions in the closed computation, but in addition the state of FSM_P is advanced according to event on the transition – note that the case of absence of a FSM_P transition corresponding to the LTSA label in induced by the computation transition corresponds to a bad activity error. In addition to these transitions, there are transitions corresponding to available transitions in the finite state in the provision computation – these represents methods calls performed by an environment. In the case of such transition, new stack is added which represents the reaction on the method call invoked by an environment. The initial computation state is identical to the one of closed computation extended with the initial state of the finite state machine.

The provision-driven computation is then represented by a special state transition system – its states correspond to the computation states and the transition relation to the available transitions. The initial state of the system is the initial computation state. There is a set of *imperative final states* which are the provision driven computation states that correspond to the end of the model computation – all stacks are empty. Additionally, there is a set of *final states* which are such imperative final states that the state of the provision finite state machine is a final state. Note that in an imperative final state the computation may continue by an action of environment.

Within the provision-driven computation, there may be identified states corresponding to a bad activity error (let E^{BA} denote them) and stuck traces (a state s which contains at least one non-empty stack will never be modified in future, i.e. the stack is identical in all the states reachable from s, let E^{stuck} denote such states). Also states corresponding to endless loop without any externally observable activity (let E^{∞} denote such states) can be identified as states from which there is no path to any final state nor any transition labelled with any externally observable event (i.e. parametrized events over $\Sigma_{prov} \cup \Sigma_{req}$). But existence of such states in itself does not mean that the model is useless. In some environments such states can be avoided.

An *observation projection*, which is also an state transition system, is then defined over the provision computations. It accumulates states of the provision computation into a super-state which hide internal non-determinism of the model in a pessimistic way. If an state of computation belongs to a super-state then also its $\tau - closure$ belongs to the super state. $\tau - closure$ of a set of states S is defined as a set of all states that

are reachable by any sequence of externally invisible transitions from any state of S. The initial state is $\tau - closure$ of the initial computation state. The transitions between the super-states are labelled with externally observable events. There is a transition labelled with an event over required method e leading from s to s' if there is at least one transition over e from a sub-state of s to a sub-state of s'. On the other hand, there is a transition from s to s' labelled with a provided method event e if for each sub-state of s there is a transition over e pointing to a sub-state of s'.

Similarly to the provision driven computation there are imperatively final states (a super-state containing at least one imperatively final sub-state) and final states (a super-state containing just final states) defined. Moreover, the set of error states is also defined in a pessimistic way, which means that if at leas one sub-state is erroneous then the super-state is erroneous. Let E^{BA} denote the set of the super-states that correspond to bad-activity and E^{NA} the set of states containing a state from E^{stuck} or E^{∞} – no activity.

For the following definition, let $!m$ denote an event over a required method m ($m \in \Sigma_{req}$) and let $?m$ denote an event over a provided method m ($m \in \Sigma_{prov}$).

Definition 2.13 (Parametrized alternation simulation)**.** Let I_{OP} and S_{OP} be observation projections of TBP models I and S. Let $Q_{I_{OP}}$ and $Q_{I_{OP}}$ denote the sets of states and $\delta_{I_{OP}}$, $\delta_{S_{OP}}$ denote the transition functions of the respective observation projections. Let P be a relation $P \subseteq Q_{I_{OP}} \times Q_{S_{OP}}$. Then a relation $\preceq_P \subseteq Q_{I_{OP}} \times Q_{S_{OP}}$ is a *parametrized alternation simulation* (with parameter P) if and only if

- $\forall (s_I, s_S) \in \preceq_P$: $(s_I, s_S) \in P$
- $\forall (s_I, s_S) \in \preceq_P$: $\delta_{I_{OP}}(s_I, !m) = s'_I \Rightarrow \; \exists s_S : \delta_{S_{OP}}(s_S, !m) = s'_S \wedge s'_I \preceq_P s'_S$
- $\forall (s_I, s_S) \in \preceq_P$: $\delta_{I_{OP}}(s_I, ?m) = s'_I \Rightarrow \; \exists s_S : \delta_{S_{OP}}(s_S, !m) = s'_S \wedge s'_I \preceq_P s'_S$

The parametrized alternation simulation, a variant of alternation simulation originally introduced for interface automata [dAH01], is extended to the observation relations in the following way:

$$I_{OP} \preceq_P S_{OP} \iff s_0^{I_{OP}} \preceq_P s_0^{S_{OP}}$$

where $s_0^{I_{OP}}$, $s_0^{S_{OP}}$ denotes the initial states of I_{OP} and S_{OP} respectively. Then I refines S with respect to P if $I_{OP} \preceq S_{OP}$. The P parameter determines which type of error the alternation simulation and in turn the refinement relation captures.

For illustration, let BA be a predicate s.t. $BA\,(a,b)$ iff $a \in E_{BA}^{I_{OP}} \Rightarrow \in E_{BA}^{S_{OP}}$ where $E_{BA}^{I_{OP}}$ is the set of bad activity states of I_{OP} and $E_{BA}^{S_{OP}}$ is the set of bad activity states of I_{OP}. Then, as it is shown in [Poc10], $\preceq_{BA}$ captures the bad activity error and $I_{OP} \preceq_{BA} S_{OP}$ implies that I refines S with respect to bad activity.

In order to define the predicate for the no activity error, it is necessary to add more information about terminating states to the the observation relation. As this work do not focus on the no activity error, the appropriate predicate will not be discussed in detail with the note that the precise definition can be found in [Poc10].

3. Analysis of Incomplete TBP Model

This work focuses on the open models with no limitation on the number of threads. More concretely, the focus is placed on a subclass of open models in which the provided interfaces that are supposed to be used from an environment are specified by provisions with full reentrancy. These models can induce massively parallel environments. The work then attempts to provide an approach to checking such highly parallel models by using counter abstraction symmetry reduction technique.

3.1 Incomplete TBP Model

From the point of view of CBD, the incomplete model corresponds to a composed component with some provided interfaces that allow unlimited re-entrant calls. The internal architecture of the component is closed except for the provided and required interfaces of the closing component which are delegated to some internal components. The provided and required interfaces form communication channels to the outside world, an environment which is not known at the current stage of the development. However, there are some requirements placed on the provided interfaces, specifically allowed call sequences. On the other hand, there might also be some known properties of the component in the form of restrictions of the possible call sequences on the required interfaces that the component may invoke. Then it is natural to ask whether the component will work correctly under any environment which fulfills the requirements placed on the provided interfaces and whether the component actually satisfies the restrictions placed on the required interfaces.

For instance consider a composed component which provides a transactional access to data records. A simplified internal architecture of such a component could for example consist of *Storage*, *Transaction Manager* and *Data Provider* components as depicted on Figure 3.1. The *Data Provider* component provides methods for reading and writing the data and methods for starting and ending transaction and the component allows the environment to call the methods in a fully re-entrant way. For example such a provision could be `(begin();(read()+write())*;(commit()+rollback()))|*`. Conversely, the *Storage* component provides reading and writing in a sequential way. The appropriate provision could then be `(read()+write())*`. The *Transaction Manager* component is then supposed to provide means for creating schedules for the transactions and utilizes the *Storage* for writing logs. If such a composition is supposed to provide an universal data access platform intended to be used in many applications then it is desirable to verify whether it will eventually work correctly under any environment which fulfills the above assumptions.

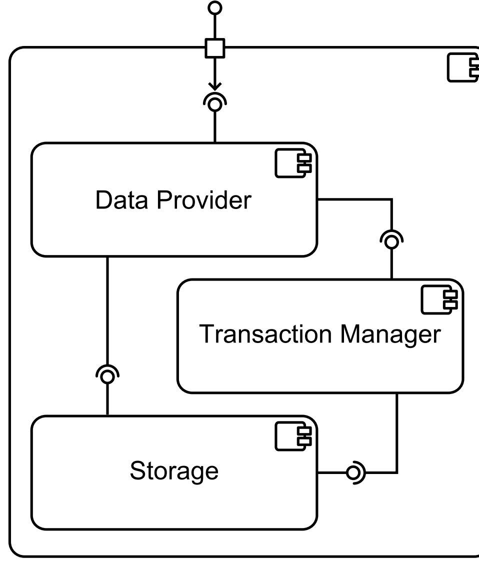

Figure 3.1: Example of a component architecture corresponding to an incomplete model.

In terms of threaded behavioral protocols, these concepts in general form an open model which is specific in the restriction that any enclosing environment can only operate on a subset of the provided methods. Recall that the composition of TBP models still provides all the methods that were provided in the subcomponents. Thus there is a need to distinguish between provided methods that are intended to be used just internally within the composition and those that will be available to the environment. Moreover, it is also necessary to separate the provisions in an analogous way. Thus, while keeping the theoretical framework defined in previous sections, a provided method can be marked as *bound* if there is a matching required method in the composition and *unbound* otherwise.

In turn, a provision Q is partially bound if at least one method which is captured by the provision (i.e. a method $m \in filter^{Q}$) is bound and the provision is fully bound if all the captured methods are bound. On the other hand, the provision is unbound if no method captured by the provision is bound. Then the methods that are captured by an unbound provision will be treated as intended to be called from the environment. A model with at least one unbound provision will be called *inclomplete* model as stated in Definition 3.1.

Definition 3.1 (Incomplete TBP Model). Let $A = ((\Sigma_{prov}, \Sigma_{req}, \Sigma_{int}), P, R, T, G)$ be a TBP model. Let $\Sigma_{bound} \subset \Sigma_{prov}$ be a set of the bound provided methods.

$P_i \in P$ is an *unbound* provision iff $filter^{P_i} \cap \Sigma_{bound} = \emptyset$, P is an *partially bound* provision iff $filter^{P_i} \cap \Sigma_{bound} \neq \emptyset$ and P is *fully bound* provision iff $filter^{P_i} \subseteq \Sigma_{bound}$.

A is *incomplete TBP model* iff all the following hold

- there are no required methods; $\Sigma_{req} = \emptyset$
- there is at least one unbound provision in P
- for arbitrary two unbound provisions $Q, R \in P$ holds that either $Q = R$ or $filter^Q \cap filter^P = \emptyset$.

The difference between incomplete and closed model is that the environment is supposed to invoke methods of unbound provisions and thus the environment is also a source of activity in the former, whereas in the closed model the only source of activity are threads of the model and the environment can not communicate with it at all. Note that the incomplete model does not require any methods from the environment. Although it is fully legitimate to require methods from the environment, it is formally not allowed in the incomplete models since the environment is not known. However, the required interfaces may be satisfied with fictive components that would assign empty reactions to the required methods. These fictive components are supposed to be supplied by the developer of the model which gives him an opportunity to specify additional restrictions on the way that the methods are called using additional provisions. On the other hand, if there are no such requirements, nothing prevents the reactions from being generated automatically. It could also be very interesting to ask whether it is possible to extract the provisions that form restrictions on the required methods in an at least semi-automatic way, but it is beyond the scope of this work.

3.1.1 Restricting Provision Expressions

This work approaches the analysis of the incomplete model by using an artificial environment which simulates any real environment that uses the unbound provided methods correctly. The artificial environment will be generated from the unbound provisions so that it will eventually call the provided methods in all the ways allowed in the provisions. In particular, it means that the environment will eventually generate all the traces specified by the provisions. This environment then closes the incomplete model. The composition of the artificial environment and the incomplete model will then form a closed model which does not allow any of its methods to be called from outside. Then the closed model will be checked as to whether it is correct with respect to *bad activity*. But since the closed model can contain an unlimited number of threads, as any unbound provision can contain full reentrancy operator, it cannot be analysed as a classical closed model. In fact, the case of an unlimited number of threads is what this work is focused on. The analysis of such a closed model will utilize symbolic counter abstraction, which reduces the symmetries induced by parallel threads and in fact allows the analysis of models with unlimited numbers of threads.

The crucial problem is to process the provisions in their general form, both in the sense of generating the artificial environment and in the terms of checking bad activity. Although symbolic counter abstraction an allows infinite number of threads, in particular an infinite number of replicas of finite number of thread types, all other entities such as variables or number of program locations must be finite. Consider for example a very simple provision `(i.a();i.b())|*;i.c()`. It says that the sequence of calls `a` and `b` can be performed infinitely many times in parallel and then c must be called. For an environment with an ambition to eventually call all the traces specified by the provision within all its execution paths it would mean that it has to launch an arbitrary number of threads that would perform `i.a();i.b()`, wait until they finish and then perform `i.c()`. But waiting until the threads finish would require a sort of counter of living threads as a state variable which would need an infinite domain.

Additionally, for checking correctness with respect to bad activity of a provision, it is necessary to somehow encode the provision and its state (in terms of its simulation along the executed method calls) into the computation state. The computation state under symbolic counter abstraction consists of a global state and thread-local states. The number of these states be finite, but the potential lies in the fact that the thread-local states can be occupied by an unbounded number of threads. Thus it is convenient to spread the provision over the thread states.. Unfortunately, this is again not possible in general.

For the reasons outlined in the previous paragraphs, it is necessary to restrict the provision expressions. Definition 3.2 proposes such a restriction. However, the concrete reasons for this particular form of provisions will become more clear during the detailed explanation of the environment generating and provision checking presented in sections 3.2.1 and 3.2.3 respectively. Nonetheless, it will be assumed that all the unbound provisions in the incomplete model are in the form of simultaneously regular provision expression, and each bound provision either does not contain full reentrancy operator or forms a simultaneously regular provision expression.

Definition 3.2 (Simultaneously regular provision)**.** A provision expression is *entirely regular* iff it does not contain $|$, $||$, $|n$ nor $|*$ operators.

A *simultaneously regular provision* expression (SRP) is defined recursively as follows:

- P, $P|n$ and $P|*$ are SRP if P is entirely regular
- $P||Q$ is SRP if P and Q are SRP and none of them contain $|$ operator
- $P|Q$ is SRP if P and Q are SRP

Note that the provisions stated in the definitions are in fact parallel compositions of regular expressions. Let P be a simultaneously regular provision expression and

$E_1, E_2, ..., E_k$ be its maximal entirely regular subexpressions. The maximality means that E_i is either P or an argument of a parallel operator ($|$, $||$, $|n$ or $|*$). Then any finite sequence $s \in traces^P$ can be taken as a parallel interleaving of a certain number of sequences, where each of them matches at least one of the entirely regular subexpressions. This fact is used in detecting whether the closed model does not contain an internal communication error with respect to bad activity. Simply put, the threads of the model will be checked as to whether they issue method calls with respect to the entirely regular subexpressions. This can be checked from the thread-local perspective. It will then be checked from the global perspective whether the threads together issue a sequence matching the provision in total. As will be shown, there are certain limits on the number of threads issuing traces of particular subexpressions that assure that the parallel interleaving will match the overall provision.

Indeed, the requirement imposed on every thread of the model, that it must satisfy a regular subexpression or issue no call in $filter^P$, as a necessary condition for validity of the model with respect to bad activity on P is stricter than in the closed model analysis. Thus the analysis of an incomplete model can contain spurious errors in the meaning, that even if the incomplete model analysis yields a discovered bad activity error, the model can still be error free for all valid environments. On the other hand, it is safe – if the incomplete analysis does not find a communication error then the model will surely not contain any bad activity error under any possible environment which satisfies the unbound provisions.

Note also, that a provision which does not contain full reentrancy can be checked using a common finite state machine which observes the events globally. Such error detection does not induce spurious errors and still remains safe as it is very close to the approach of checking bad activity in the closed model. Moreover, some potential may lie in a combination of these two techniques in order to fully check re-entrant provisions and reduce spurious errors induced by the former.

3.2 Analysis Using Symbolic Counter Abstraction

In order to check whether an incomplete TBP model is correct with respect to the bad activity, several transformations need to be done on the model.

The unbound provisions need to be transformed to an artificial component that will substitute any environment which act correctly with respect to the provisions. This component is supposed to call methods of the unbound provisions in a valid and exhaustive manner – just all traces of the provisions must be eventually performed. The artificial component, in the case of full re-entrant provisions, consists of an unlimited number of threads. However, these threads arise from unbounded replication of a finite number of thread types. Then the composition of the artificial component with

the components corresponding to the incomplete model forms a closed model. Such a closed model is in fact a replicated finite state model and thus the counter abstraction is applicable on it.

As already described in chapter 1, the counter abstraction reduces symmetries induced by the replication by reducing the thread state reachability problem to the VASS coverability problem. This reduction is done by using counters which capture the numbers of threads residing in individual thread states.

In order to use a TBP model with the explicit counter abstraction, it would be necessary to convert the model's imperative parts, represented by LTSAs, to VASS. However, as the symbolic counter abstraction interleaves the model checking phase with the state space exploration phase, it is only necessary to present a way in which the exploration phase will be performed on LTSAs. Nonetheless, in order to clarify the exploration phase and the relation between LTSA and appropriate VASS, even the explicit conversion will be outlined in Section 3.2.2.

Finally, the bound provisions need to be encoded into the model states. More precisely, the properties of the model induced by the provisions need to be mapped onto the reachability of thread states and in turn converted to the coverability of VASS configurations. This issue is addressed in section 3.2.2.

3.2.1 Artificial Environment from Unbound Provisions

The goal of this section is to design a mechanism of transforming a provision to an artificial component. The component will act as a substitute for any arbitrary component behind the provision, that respects the provision. Hence the artificial component must perform only such sequences of method calls, that belong to the traces specified by the provisions. Moreover, considering all possible executions of the component, all traces of the provision must be invoked. It is obvious, that the component may contain several thread instances, even possibly unlimited number in case that the provision contains full reentrancy operator, in order to involve the parallel nature of the provision. The desirable result can be accomplished using systematic coordination of the threads and using non-determinism within them.

Before the particular transformation can be presented, a more detailed theoretical analysis of the provision expressions is needed. It will offer arguments for the claims above and shape the transformation itself.

Provision Expression Analysis

Considering a general provision expression,he and-parallel operator is defined via interleavings of operands and the or-parallel and reentrancy operators are based on it. The interleaving is expressed formally in definition 3.3.

Definition 3.3 (Interleaving)**.** Let $\alpha, \beta \in \Sigma^*$ be finite sequences of symbols of Σ (i.e. words over Σ). Let $|\alpha|$ denote length of α, α^i denote i-th symbol of α.

We say that $\gamma \in \Sigma^*$ is an *interleaving* of α and β iff $|\gamma| = |\alpha| + |\beta|$ and $\exists \varphi_\alpha : \{1,2,...,|\alpha|\} \to \{1,2,...,|\gamma|\}, \varphi_\beta : \{1,2,...,|\beta|\} \to \{1,2,...,|\gamma|\}$ strictly increasing functions such that $\forall i \in \{1,2,...,|\alpha|\}, j \in \{1,2,...,|\beta|\} : \varphi_\alpha(i) \neq \varphi_\beta(j),\ \alpha^i = \gamma^{\varphi_\alpha(i)}$ and $\beta^j = \gamma^{\varphi_\beta(j)}$.

Let $A, B \subseteq \Sigma^*$ be finite sets of words over Σ, then $A \otimes B$ denotes a set of *mutual interleavings* of elements of A and B. Formally, $\gamma \in A \otimes B \iff \exists \alpha \in A, \beta \in B$ s.t. γ is interleaving of α and β.

Following notation will also be used ($k \in \mathbb{N}_0$, $m, n \in \mathbb{Z}, n \leq m$):

$$A^{\otimes k} = \underbrace{A \otimes A \otimes \ldots \otimes A}_{k}$$

$$\bigotimes_{i=n}^{m} A_i = A_n \otimes A_{n+1} \otimes \ldots \otimes A_{m-1} \otimes A_m$$

The $\otimes$ operator is clearly *commutative*. Directly from definition also follows that $A \otimes \emptyset = \emptyset$ and $A \otimes \{\lambda\} = A$ where λ denotes an empty word. It is also clear that $A^{\otimes(k+l)} = A^{\otimes k} \otimes A^{\otimes l}$ for any $k, l \in \mathbb{N}_0$. Thus $A^{\otimes 0} = \{\lambda\}$ for $A \neq \emptyset$ as $A^{\otimes k} = A^{\otimes k} \otimes A^{\otimes 0}$ and clearly $\{\lambda\}$ is the only identity element with respect to $\otimes$. Note that $\emptyset^{\otimes 0}$ is also equal to $\{\lambda\}$.

Having two provision expressions P and Q, the semantics of the $|$ and $||$ operators can be expressed in the following way:

$$traces^{P|Q} = traces^{P} \otimes traces^{Q} \tag{3.1}$$

$$traces^{P||Q} = \left(traces^{P} \otimes traces^{Q}\right) \cup traces^{P} \cup traces^{Q} \tag{3.2}$$

Lemma 3.4 (Distributivity)**.** *Let $A, B, C \subseteq \Sigma^*$ be sets of finite words over Σ. Then*

$$(A \cup B) \otimes C = (A \otimes C) \cup (B \otimes C)$$

Proof.

$$\begin{aligned}
\gamma \in (A \cup B) \otimes C &\iff (\exists \alpha \in A \cup B)(\exists \beta \in C) : \gamma \text{ interleaving of } \alpha, \beta \\
&\iff (\exists \alpha \in A)(\exists \beta \in C) : \gamma \text{ interleaving of } \alpha, \beta \\
&\qquad \vee \\
&\qquad (\exists \alpha \in B)(\exists \beta \in C) : \gamma \text{ interleaving of } \alpha, \beta \\
&\iff (\gamma \in A \otimes C) \vee (\gamma \in B \otimes C) \\
&\iff \gamma \in (A \otimes C) \cup (B \otimes C)
\end{aligned}$$

$\square$

Considering that $P|n$ is defined as $P||P||...||P$, where P occurs n-times on the right side and $P|*$ is $P||P||...$, following inclusions are direct consequence of repeated application of Lemma 3.4.

$$\left(traces^P\right)^{\otimes k} \subseteq traces^{P|n} \quad \forall k \leq n; k,n \in \mathbb{N} \tag{3.3}$$

$$\left(traces^P\right)^{\otimes k} \subseteq traces^{P|*} \quad \forall k \in \mathbb{N} \tag{3.4}$$

So far, the general provision expressions have been considered. Now the simultaneously regular provisions will be focused. Such expression consists of entirely regular subexpressions. Considering only maximal regular subexpressions, i.e. such that they do not act as an operand of a regular expression operator, they can be enumerated for example in order of appearance in the whole expression from left to right. Note that the subexpressions are not necessarily unique, there may be the same subexpressions as they appear more than once.

Let P be a simultaneously regular provision expression consisting of $E_1, E_2, ..., E_k$ maximal entirely regular subexpressions which define regular languages $R_1, R_2, ..., R_k$. Indeed, not all the words of $R_1^{\otimes i_1} \otimes R_2^{\otimes i_2} \otimes ... \otimes R_k^{\otimes i_k}$ for all possible vectors $(i_1, i_2, ..., i_k) \in (N_0)^k$ must match the provision P. But there are some valid vectors which induce that the interleavings match P, these will be identified in the following.

First, consider the maximal number of the words that match a particular entirely regular subexpression. Clearly there may be a certain number, which when exceeded, the interleaving will not match the provision. These numbers will be called *upper parallel boundaries*. E.g. for the $(\mathtt{a}+\mathtt{b})|n$ expression, the upper parallel boundary is n. On the other hand, there is no such limit for the expression $(\mathtt{a}+\mathtt{b})|*$, the number of the words matching $\mathtt{a}+\mathtt{b}$ may be arbitrarily large. Definition 3.5 states prescription for the upper parallel boundaries. Subsequently, Observation 3.6 covers the previous ideas formally.

Definition 3.5 (Upper parallel boundaries)**.** Let P be a simultaneously regular provision expression and A be a subexpression of P.

$\mathcal{U}_P(A)$ is an *upper parallel boundary* of A in P and it is defined as follows:

- $\mathcal{U}_P(A) = 1$ if $A = P$
- $\mathcal{U}_P(A) = n \cdot \mathcal{U}_P(B)$ if B is a subexpression of P such that $B = A|n$;
- $\mathcal{U}_P(A) = \omega$ if $A|*$ is also a subexpression of P;
- $\mathcal{U}_P(A) = \mathcal{U}_P(B)$ if B, C are subexpressions of P s.t. $B = A*$ or $B = A \circ C$ where $\circ$ stands for $|$, $||$, $+$ or ;.

Note that from the last point follows that $\mathcal{U}_P(A) = \mathcal{U}_P(B)$ if A, B are subexpressions of P, A is a subexpression of B and there is no such subexpression C of B that A is a subexpression of C and $C|n$ or $C|*$ is also a subexpression of B.

Observation 3.6 (Upper parallel boundaries and interleaving)**.** *Let P be a simultaneously parallel provision expression consisting of $E_1, E_2, \dots, E_k$ entirely regular expressions and $\mathcal{U}_P(E_i)$ be the upper parallel boundary for E_i in P. Having a vector $\eta \in \mathbb{N}^k$ such that $\eta_{(i)} \leq \mathcal{U}_P(E_i)$ for all the i, then*

$$\bigotimes_{i=1}^{k} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}} \subseteq traces^P$$

Intuitively, on the opposite side of the scale, there will also be a sort of boundaries. On closer inspection, it is crucial whether at least one word matching certain regular subexpression is mandatory. Unfortunately, this can not be expressed independently of other subexpressions. Consider $(\mathtt{a}+\mathtt{b})||(\mathtt{c};\mathtt{d})$ provision expression. A word matching $(\mathtt{a}+\mathtt{b})$ is mandatory just in the case that there is no word matching $(\mathtt{c};\mathtt{d})$. Intuitively, for the $(\mathtt{a}+\mathtt{b})||(\mathtt{c};\mathtt{d})$ example, the relation between its subexpression can be expressed as "$(\mathtt{a}+\mathtt{b})||(\mathtt{c};\mathtt{d})$ will be met if $(\mathtt{a}+\mathtt{b})$ is met or $(\mathtt{c};\mathtt{d})$ is met". The lower boundaries on the number of words are articulated as propositional expressions.

Definition 3.7 (Lower parallel guards)**.** Let P be a simultaneously regular provision expression.

Then *lower parallel guard* of P denoted for as $\mathcal{L}_P$ is a propositional formula defined as follows:

- $\mathcal{L}_P = \mathcal{L}_A \wedge \mathcal{L}_B$ if $P = A|B$
- $\mathcal{L}_P = \mathcal{L}_A \vee \mathcal{L}_B$ if $P = A||B$
- $\mathcal{L}_P = \mathcal{L}_A$ if $P = A|n$ or $P = A|*$
- $\mathcal{L}_P = \sigma_P$ if P is an entirely regular provision expression; σ_P is a propositional variable.

Note that σ_P would informally correspond to "P is met". This specifically means, that there is at least one word matching P in the overall interleaving. The following theorem summarizes the properties of defined boundaries. In other words, it says that if there is a set of words that correspond to the upper boundaries and lower guards given by a provision, interleaving of these words will match the provision. Moreover, all the possible matching sets will form the provision exhaustively.

Theorem 3.8. *Let P be a simultaneously parallel provision expression consisting of $E_1, E_2, \dots, E_k$ maximal entirely regular expressions. Let $\mathcal{L}_P$ be the lower parallel guard*

of P containing the propositional variables $\sigma_{E_1}, \sigma_{E_2}, ..., \sigma_{E_k}$ and $\mathcal{U}_P(E_i)$ be the upper parallel boundary for E_i in P.

Having a vector $\eta \in (\mathbb{N}_0)^k$, let ϑ_η denote an interpretation of variables $\sigma_{E_1}, \sigma_{E_2}, ..., \sigma_{E_k}$ such that $\vartheta_\eta(\sigma_{E_i})$ is true if and only if $\eta_{(i)} \geq 1$. If F is a propositional formula over $\sigma_{E_1}, \sigma_{E_2}, ..., \sigma_{E_k}$ variables, then let also $\vartheta_\eta(F)$ denote value of formula F under ϑ_η.

Then for all $\eta \in (\mathbb{N}_0)^k$ s.t. $\forall i \in \{1,2,...,k\} : \eta_{(i)} \leq \mathcal{U}_P(E_i)$ the following holds:

$$\vartheta_\eta(\mathcal{L}_P) \text{ is } true \Longrightarrow \bigotimes_{i=1}^{k} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}} \subseteq traces^P \tag{3.5}$$

Moreover, let $\mathcal{N}$ be a set of all vectors $\eta \in (\mathbb{N}_0)^k$ such that $\vartheta_\eta(\mathcal{L}_P)$ is true and $\forall i \in \{1,2,...,k\} : \eta_{(i)} \leq \mathcal{U}_P(E_i)$. Then

$$\bigcup_{n \in \mathcal{N}} \bigotimes_{i=1}^{k} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}} = traces^P \tag{3.6}$$

Before a proof will be presented, note that the observation 3.6 is a consequence of this theorem (specifically of the (3.5) implication). Clearly a vector η such that $\eta_{(i)} \geq 1$ will conform any lower parallel guard. This advocates the assumptions of the observation to satisfy the premise of (3.5).

Proof. First, a proof of (3.5) using complete induction will be presented. The induction will be led over number of binary parallel operators ($|$ and $||$) in a provision expression.

Consider the base case – let P be a simultaneously regular provision expression without any binary parallel operator. From definition, P contains exactly one maximal entirely regular subexpression E_1. Hence the left side of inclusion in (3.5) can be simplified to $\left(traces^{E_1}\right)^{\otimes \eta_{(1)}}$ using substitution of $k = 1$. Three are three possible forms of P:

1. P is entirely regular expression, i.e. $P = E_1$. Then $\mathcal{U}_P(E_1) = 1$ and $\mathcal{L}_P = \sigma_{E_1}$, thus the only vector η such that $\vartheta_\eta(\mathcal{L}_P)$ is *true* and $\eta_{(1)} \leq \mathcal{U}_P(E_1)$ is (1). Then

$$\left(traces^{E_1}\right)^{\otimes \eta_{(1)}} = \left(traces^{E_1}\right)^{\otimes 1} = traces^{E_1} = traces^P$$

2. $P = E_1|n$ where E_1 is an entirely regular expression. Then $\mathcal{U}_P(E_1) = n$ and $\mathcal{L}_P = \sigma_{E_1}$. Hence if a vector η is such that $\vartheta_\eta(\mathcal{L}_P)$ is *true*, then $1 \leq \eta_{(1)}$. Additionally $\eta_{(1)} \leq \mathcal{U}_P(E_1) = n$ from assumption on η. From (3.3)

$$\left(traces^{E_1}\right)^{\otimes \eta_{(1)}} \subseteq traces^P$$

3. $P = E_1|*$ where E_1 is entirely regular expression. Analogously to the previous

case, $1 \leq \eta_{(1)}$ and from (3.4)

$$\left(traces^{E_1}\right)^{\otimes \eta_{(1)}} \subseteq traces^P$$

Let P contain $m > 0$ binary parallel operators, then there are Q and R simultaneously regular expressions such that either $P = Q|R$ or $P = Q||R$. Let Q (resp. R) contain $F_1, ..., F_k$ (resp. $G_1, ..., G_l$) maximal entirely regular expressions. Without loss of generality we can assume that P contains $E_1, ..., E_{k+l}$ maximal entirely regular expressions such that $E_i = F_i$ for $1 \leq i \leq k$ and $E_{k+i} = G_i$ for $1 \leq i \leq l$. Hence

$$\mathcal{U}_P(E_i) = \begin{cases} \mathcal{U}_Q(F_i) & \text{if } 1 \leq i \leq k \\ \mathcal{U}_R(G_{i-k}) & \text{if } k+1 \leq i \leq k+l \end{cases} \tag{3.7}$$

Both Q and R contain obviously less than m binary parallel operators. From induction hypothesis for all $\eta \in (\mathbb{N}_0)^k$ s.t. $\forall i : \eta_{(i)} \leq \mathcal{U}_Q(F_i)$

$$\vartheta_\eta(\mathcal{L}_Q) \text{ is } true \Longrightarrow \bigotimes_{i=1}^{k} \left(traces^{F_i}\right)^{\otimes \eta_{(i)}} \subseteq traces^Q$$

and for all $\eta \in (\mathbb{N}_0)^l$ s.t. $\forall i : \eta_{(i)} \leq \mathcal{U}_R(G_i)$

$$\vartheta_\eta(\mathcal{L}_R) \text{ is } true \Longrightarrow \bigotimes_{i=1}^{l} \left(traces^{G_i}\right)^{\otimes \eta_{(i)}} \subseteq traces^R$$

Considering the case that $P = Q|R$, let $\eta\,(\mathbb{N}_0)^{k+l}$ be a vector such that $\eta_{(i)} \leq \mathcal{U}_P(E_i)$ and $\vartheta_\eta(\mathcal{L}_P)$ is $true$. Then both $\vartheta_n(\mathcal{L}_Q)$ and $\vartheta_n(\mathcal{L}_R)$ are $true$ because $\mathcal{L}_P = \mathcal{L}_Q \wedge \mathcal{L}_R$. Then

$$\begin{aligned} \bigotimes_{i=1}^{k+l} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}} &= \left(\bigotimes_{i=1}^{k} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}}\right) \otimes \left(\bigotimes_{i=k+1}^{k+l} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}}\right) \\ &= \left(\bigotimes_{i=1}^{k} \left(traces^{F_i}\right)^{\otimes \eta_{(i)}}\right) \otimes \left(\bigotimes_{j=1}^{l} \left(traces^{G_j}\right)^{\otimes \eta_{(j+k)}}\right) \\ &\subseteq traces^Q \otimes traces^R \\ &= traces^P \end{aligned}$$

Note that the inclusion follows from monotonicity of $\otimes$ operator, which is evident.

For the case $P = Q||R$ the lower parallel guard is $\mathcal{L}_P = \mathcal{L}_Q \vee \mathcal{L}_R$. But from Def-

inition 3.2 neither Q nor R contain $|$ operator, thus both $\mathcal{L}_Q$ and $\mathcal{L}_R$ take the form of a disjunction of some propositional variables. Let $\eta \in (\mathbb{N}_0)^{k+l}$ be a vector such that $\eta_{(i)} \leq \mathcal{U}_P(E_i)$ and $\vartheta_\eta(\mathcal{L}_P)$ is *true*. Additionally, let $\eta_{(i,j)} \in (\mathbb{N}_0)^{j-i+1}$, $i < j \leq k$ denote a part of vector η beginning at its i-th coordinate and ending at j-th coordinate; $\forall 1 \leq \alpha \leq j-i+1 : \left(\eta_{(i,j)}\right)_{(\alpha)} = \eta_{(i+\alpha-1)}$. Then either both $\vartheta_\eta(\mathcal{L}_Q)$ and $\vartheta_\eta(\mathcal{L}_R)$ are *true* or just one of $\eta_{(1,k)}$, $\eta_{(k+1,k+l)}$ is zero in all its coordinates. If $\vartheta_\eta(\mathcal{L}_Q) \wedge \vartheta_\eta(\mathcal{L}_R)$ then (analogously to the previous case)

$$\bigotimes_{i=1}^{k+l} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}} \subseteq traces^Q \otimes traces^R$$

Otherwise suppose that $\eta_{(k+1,k+l)} = \vec{0}$. Then obviously

$$\begin{aligned}\bigotimes_{i=1}^{k+l} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}} &= \left(\bigotimes_{i=1}^{k} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}}\right) \otimes \left(\bigotimes_{i=k+1}^{k+l} \left(traces^{E_i}\right)^{\otimes 0}\right) \\ &= \left(\bigotimes_{i=1}^{k} \left(traces^{F_i}\right)^{\otimes \eta_{(i)}}\right) \\ &\subseteq traces^Q\end{aligned}$$

For the opposite case ($\eta_{(1,k)} = \vec{0}$) holds by analogy:

$$\bigotimes_{i=1}^{k+l} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}} \subseteq traces^R$$

Hence

$$\begin{aligned}\bigotimes_{i=1}^{k+l} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}} &\subseteq \left(traces^Q \otimes traces^R\right) \cup traces^Q \cup traces^P \\ &= traces^P\end{aligned}$$

Having (3.5) proven, the only thing rests to show is that all the elements of the right side of (3.6) belong to the left side, in other words:

$$\bigcup_{\eta \in \mathcal{N}} \bigotimes_{i=1}^{k} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}} \supseteq traces^P \tag{3.8}$$

As the opposite inclusion is a direct consequence of (3.5), (3.8) affirms (3.6).

Let us show (3.8) again using induction over the number of binary parallel operators in the provision expression. Let P contain neither $|$ nor $||$, then P contains exactly one maximal entirely regular expression E_1. Thus after substitution of k:

$$\bigcup_{\eta\in\mathcal{N}} \bigotimes_{i=1}^{k} \left(traces^{E_i}\right)^{\otimes n_i} = \bigcup_{\eta\in\mathcal{N}} \left(traces^{E_1}\right)^{\otimes\eta(1)}$$

There are three possible forms of such P:

1. P is the entirely regular expression itself, i.e. $P = E_1$, $\mathcal{L}_P = \sigma_{E_1}$, $\mathcal{U}_P(E_1) = 1$. Hence $\mathcal{N} = \{(1)\}$ and

$$\bigcup_{\eta\in\mathcal{N}} \left(traces^{E_1}\right)^{\otimes\eta(1)} = traces^{E_1} = traces^{P}$$

2. $P = E_1|n$ where E_1 is entirely regular expression $\mathcal{L}_P = \sigma_{E_1}$, $\mathcal{U}_P(E_1) = n$. Hence $\mathcal{N} = \{(1),(2),...,(n)\}$. Let $\gamma \in traces^P$, then by definition there are $\gamma_1,\gamma_2,...,\gamma_j$, $j \leq n$, such that $\forall i \in \{1,2,...j\} : \gamma_i \in traces^{E_1}$ and γ is an interleaving of $\gamma_1,\gamma_2,...,\gamma_j$. Thus $\gamma \in \left(traces^{E_i}\right)^{\otimes j}$. Obviously $(j) \in \mathcal{N}$, thus

$$\gamma \in \left(traces^{E_i}\right)^{\otimes j} \subseteq \bigcup_{\eta\in\mathcal{N}} \left(traces^{E_i}\right)^{\otimes\eta(i)}$$

3. $P = E_1|*$ where E_1 is entirely regular expression $\mathcal{L}_P = \sigma_{E_1}$, $\mathcal{U}_P(E_1) = \omega$. Hence $\mathcal{N} = \{(1),(2),...\}$. Let $\gamma \in traces^P$, then there is $j \in \mathbb{N}$ such that $\gamma \in \left(traces^{E_1}\right)^{\otimes j}$ analogously to the previous case. Clearly $(j) \in \mathcal{N}$, thus

$$\gamma \in \left(traces^{E_i}\right)^{\otimes j} \subseteq \bigcup_{\eta\in\mathcal{N}} \left(traces^{E_i}\right)^{\otimes\eta(i)}$$

Let P contain m binary parallel operators, $m > 0$. Thus $P = Q|R$ or $P = Q||R$ where Q and R are simultaneously regular provision expressions containing less than m binary parallel operators. Let Q (resp. R) contain $F_1,...,F_k$ (resp. $G_1,...,G_l$) maximal entirely regular expressions. Without loss of generality P contains $E_1,...,E_{k+l}$ maximal entirely regular expressions such that $E_i = F_i$ for $1 \leq i \leq k$ and $E_{k+i} = G_i$ for $1 \leq i \leq l$. By the induction hypothesis

$$\bigcup_{\eta\in\mathcal{N}^Q} \bigotimes_{i=1}^{k} \left(traces^{F_i}\right)^{\otimes\eta(i)} \supseteq traces^{Q}$$

and

$$\bigcup_{\eta \in \mathcal{N}^R} \bigotimes_{i=1}^{l} \left(traces^{G_i}\right)^{\otimes \eta_{(i)}} \supseteq traces^R$$

In the case of $P = Q|R$

$$\begin{aligned}
traces^P &= traces^Q \otimes traces^R \\
&\subseteq \left(\bigcup_{\eta \in \mathcal{N}^Q} \bigotimes_{i=1}^{k} \left(traces^{F_i}\right)^{\otimes \eta_{(i)}} \right) \otimes \left(\bigcup_{\eta \in \mathcal{N}^R} \bigotimes_{i=1}^{l} \left(traces^{G_i}\right)^{\otimes \eta_{(i)}} \right) \\
&\subseteq \bigcup_{\eta \in \mathcal{N}^P} \bigotimes_{i=1}^{k+l} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}} \qquad (3.9)
\end{aligned}$$

where the last inclusion (3.9) is left to be proven. Let $\gamma \in traces^P$, there are α, β such that $\gamma = \alpha \otimes \beta$ and

$$\alpha \in \bigcup_{\eta \in \mathcal{N}^Q} \bigotimes_{i=1}^{k} \left(traces^{F_i}\right)^{\otimes \eta_{(i)}}$$

$$\beta \in \bigcup_{\eta \in \mathcal{N}^R} \bigotimes_{i=1}^{l} \left(traces^{G_i}\right)^{\otimes \eta_{(i)}}$$

thus there are $\eta^q \in \mathcal{N}^Q$ and $\eta^r \in \mathcal{N}^R$ such that

$$\alpha \in \bigotimes_{i=1}^{k} \left(traces^{F_i}\right)^{\otimes \eta^q_{(i)}} \text{ and } \beta \in \bigotimes_{i=1}^{l} \left(traces^{G_i}\right)^{\otimes \eta^r_{(i)}}$$

Let $\eta^{qr} \in (\mathbb{N}_0)^{k+l}$ be defined as a concatenation of η^q and η^r:

$$\eta^{qr}_{(i)} = \begin{cases} \eta^q_{(i)} & \text{if } 1 \leq i \leq k \\ \eta^r_{(i-k)} & \text{if } k+1 \leq i \leq k+l \end{cases}$$

The concatenation is suitable to generate γ from $E_1, E_2, ..., E_{k+l}$:

$$\gamma \in \bigotimes_{i=1}^{k+l} \left(traces^{E_i}\right)^{\otimes \eta^{qr}_{(i)}}$$

But $\eta^{qr} \in \mathcal{N}^P$ because of (3.7) and $\mathcal{L}_P = \mathcal{L}_Q \wedge \mathcal{L}_R$ (expressions hold for both η^q and η^r, thus the conjunction holds for the concatenation η^{qr}), which proves the (3.9) inclusion.

For the case that $P = Q||R$ holds

$$traces^P = \left(traces^Q \otimes traces^R\right) \cup traces^Q \cup traces^R$$

Let $\gamma \in traces^P$. The proof of

$$\gamma \in traces^Q \otimes traces^R \Longrightarrow \gamma \in \bigcup_{\eta \in \mathcal{N}^P} \bigotimes_{i=1}^{k+l} \left(traces^{E_i}\right)^{\otimes \eta_{(i)}}$$

is analogous to the previous case where P was $Q|R$. The only difference is that now $\mathcal{L}_P$ is $\mathcal{L}_Q \vee \mathcal{L}_R$, which is even less restrictive, thus the concatenated sequence will belong to $\mathcal{N}^P$ as well.

If $\gamma \in traces^Q$ then there is $\eta^q \in \mathcal{N}^Q$ such that $\omega \in \bigotimes_{i=1}^{k} \left(traces^{F_i}\right)^{\otimes \eta^q_{(i)}}$. Let $\eta^{q0} \in (\mathbb{N}_0)^{k+l}$ be defined as an extension of η^q:

$$\eta^{q0}_{(i)} = \begin{cases} \eta^q_{(i)} & \text{if } 1 \le i \le k \\ 0 & \text{if } k+1 \le i \le k+l \end{cases}$$

The extension generates γ as well, i.e. $\omega \in \bigotimes_{i=1}^{k+l} \left(traces^{E_i}\right)^{\otimes \eta^{q0}_{(i)}}$. Since $\mathcal{L}_Q$ holds under interpretation induced by η^q, regardless $\mathcal{L}_R$, $\mathcal{L}_P = \mathcal{L}_Q \vee \mathcal{L}_R$ holds under interpretation induced by η^{q0}. The upper parallel boundaries are unchanged according to (3.7), so $\eta^{q0} \in \mathcal{N}^P$. The case $\gamma \in traces^R$ is completely analogous. $\square$

Provisions to Imperative Code

So far, the rules for interleaving words of entirely regular subexpressions of a simultaneously regular provision expression has been stated. Now it is convenient to think of creating imperative code that would generate the words of the entirely regular subexpressions – in terms of method calls. The code will reside in several threads in order to obtain interleaving of call sequences by interleaving of the threads' runs. On the other hand, non-deterministic choices will allow generating all the possible call sequences corresponding to a regular expression.

At first glance there is a simple way in which the individual threads are specialized for generating the sequences of events for the individual regular expressions. For each regular subexpression there would be a thread *type*, that would correspond to it. The number of replicas of the thread types would reflect the upper parallel boundaries. But this approach soon gets into trouble with return values of method calls. E.g. for $(\texttt{a}() : \texttt{x}; \texttt{b}())||(\texttt{a}() : \texttt{y}; \texttt{c}())$ provision, there would be two threads; one expects that a

returns x, the other y. If the first thread retrieves y as the return value, which is not necessarily a violation of the provision, the thread is not able to cope with it. An intuitive solution would be to *merge* the thread types to accept both return values.

Previous reasoning leads to a more universal approach, where there is a single thread type that is able to perform all the sequences matching the union of the entirely regular subexpressions. The thread should also be able to distinguish which particular regular expressions are satisfied along the generated call sequences. For the sake of clarity a finite state machine that accepts union of regular languages given by entirely regular subexpressions will be defined. The definition 3.9 follows standard disjunction of finite state machines and consequent determinization. The relation to the original finite state machines is captured formally by defining resulting states as sets of original states.

Definition 3.9. Let P be a simultaneously regular provision expression, and $E_1, E_2, ..., E_k$ be the maximal entirely regular subexpressions of P. Let $\mathcal{A}_i$ denote a deterministic finite state machine representing E_i. Without loss of generality let $\forall i, j \in \{1, 2, ..., k\}$: $Q_{\mathcal{A}_i} \cap Q_{\mathcal{A}_j} = \emptyset$.

Then $\mathcal{D}_P = (Q, \Sigma, \delta, s_0, F)$ is a finite state machine such that

- the set of states $Q \subseteq \wp\left(\bigcup_{i=1}^{k} Q_{\mathcal{A}_i}\right)$;
- the alphabet $\Sigma = \bigcup_{i=1}^{k} \Sigma_i$, where Σ_i denotes alphabet (input symbols) of $\mathcal{A}_i$;
- the transition relation δ is defined as follows:

$$
\begin{aligned}
(S, x, T) \in \delta \quad \iff \quad & S \neq \emptyset \wedge T \neq \emptyset \\
\wedge \quad & \forall s \in S \; \forall i \in \{1, 2, ..., k\} \; \forall t \in Q_{\mathcal{A}_i} : \\
& (s, x, t) \in \delta_{\mathcal{A}_i} \Rightarrow t \in T \\
\wedge \quad & \forall t \in T \; \exists s \in S \; \exists i \in \{1, 2, ..., k\} : \\
& (s, x, t) \in \delta_{\mathcal{A}_i}
\end{aligned}
$$

- the initial state s_0 is the set of the initial states of $\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_k$;
- the set of final states $F \subseteq Q$ is defined as follows:

$$S \in F \iff \exists s \in S \; \exists i \in \{1, 2, ..., k\} : s \in F_{\mathcal{A}_i}$$

For instance, if a simulation over $\mathcal{D}_P$ reaches the state $\{a, b\}$ where $a \in \mathcal{A}_1$ and $b \in \mathcal{A}_2$, then the simulation of the same word over $\mathcal{A}_1$ and $\mathcal{A}_2$ would end in the states a and b respectively. Let us say that a state p of $\mathcal{D}_P$ *overlaps* with $\mathcal{A}_i$ and in turn with E_i if there is at least one state of $\mathcal{A}_i$ which is also an element of p. Then it is convenient to define function $\mathcal{C}_P$ as a function which maps each state of $\mathcal{D}_P$ to a set of indexes of the original

state machines that the state overlaps with. More formally, let P contain k maximal entirely regular subexpressions $E_1, E_2, ..., E_k$ which correspond to deterministic finite state machines $\mathcal{A}_1, \mathcal{A}_2, ...\mathcal{A}_k$. The function $\mathcal{C}_P : Q_{\mathcal{D}_P} \to \wp(\{1,2,...,k\})$ can be defined as

$$\mathcal{C}_P(p) = \{i \mid 1 \le i \le k \wedge \exists q \in Q_{\mathcal{A}_i} : q \in p\} \quad \forall p \in Q_{\mathcal{D}_P} \tag{3.10}$$

Moreover, let $\mathcal{F}_P : Q_{\mathcal{D}_P} \to \wp(\{1,2,...,k\})$ be a function defined as

$$\mathcal{F}_P(p) = \{i \mid 1 \le i \le k \wedge \exists q \in F_{\mathcal{A}_i} : q \in p\} \quad \forall p \in Q_{\mathcal{D}_P} \tag{3.11}$$

where $F_{\mathcal{A}_i}$ denotes the set of final states $\mathcal{A}_i$.

Lemma 3.10 (Specialization). *Let P be a simultaneously regular provision expression and let Q, δ denote the set of states of $\mathcal{D}_P$ and the transition relation of $\mathcal{D}_P$ respectively.*

Then for every $p, q \in Q$

$$(\exists x : (p,x,q) \in \delta) \Rightarrow \mathcal{C}_P(q) \subseteq \mathcal{C}_P(p)$$

Proof. Let $E_1, E_2, ..., E_k$ be the maximal entirely regular subexpressions of P and $\mathcal{A}_1$, $\mathcal{A}_2, ..., \mathcal{A}_k$ be the deterministic finite state machines according to Definition 3.9. Let $p, q \in Q_{\mathcal{D}_P}$ s.t. $p \neq \emptyset$ and $q \neq \emptyset$ and there is a transition from p to q. Finally, let us assume that $\mathcal{C}_P(q) \not\subseteq \mathcal{C}_P(p)$ for contradiction.

Then there is j s.t. $j \in \mathcal{C}_P(q)$ and $j \notin \mathcal{C}_P(p)$. From $j \in \mathcal{C}_P(q)$ follows that there is a state $q_j \in Q_{\mathcal{A}_j}$ s.t. $q_j \in q$. On the other hand, $p \cap Q_{\mathcal{A}_j} = \emptyset$ is a consequence of $j \notin \mathcal{C}_P(p)$. From definition of the transition relation of $\mathcal{D}_P$, there is no transition from p to q since there is no such $r \in p$ that there is a transition from r to q_j in $\mathcal{A}_j$.

But the absence of transition from p to q is in contradiction with the assumptions.

□

Note that the method of construction of $\mathcal{D}_P$ suggests that (1) the number of overlapped subexpressions does not grow along the transitions (from Lemma 3.10), and (2) if a state overlapping with some subexpressions is reached on the path accepting a particular word, the word would also be accepted by at least one state machine representing these subexpressions, i.e. the word matches at least one of the subexpressions. The second statement follows from (1) and the obvious fact that $\mathcal{F}_P(p) \subseteq \mathcal{C}_P(p)$ for any p.

A LTSA fragment which generates method calls that form words accepted by $\mathcal{D}_P$ will structurally correspond to the $\mathcal{D}_P$ machine. Its set of states will be derived from $Q_{\mathcal{D}_P}$.

$\mathcal{D}_P$ transitions for method call events will be represented by transitions with the particular method calls as labels. Transitions for return events will be represented as transitions with guards checking the return value. The following paragraph prescribes the relation between $\mathcal{D}_P$ and the LTSA more precisely.

Let P be a simultaneously regular provision expression, $\mathcal{D}_P$ be a finite state machine according to Definition 3.9. LTSA generating exactly the words accepted by $\mathcal{D}_P$, denoted $LTSA_{\mathcal{D}_P}$, is created as follows

- States of $LTSA_{\mathcal{D}_P}$ correspond to the states of $\mathcal{D}_P$ and contain three additional states s, f and e which respectively denote the initial state, the final state and the error state (structurally, the last is also a final state). More precisely, let $S_{\mathcal{D}_P}$ be the set of states of $\mathcal{D}_P$, $S_{LTSA_{\mathcal{D}_P}}$ be the set of states of $LTSA_{\mathcal{D}_P}$. Then there is a set $S \subset S_{LTSA_{\mathcal{D}_P}}$ such that $S_{LTSA_{\mathcal{D}_P}} = S \cup \{s, f, e\}$ and there is a bijection $\ell_P : S_{\mathcal{D}_P} \to S$.

- Transitions of $LTSA_{\mathcal{D}_P}$ also correspond to the transitions of $\mathcal{D}_P$. Let $t = (s, x, t) \in \delta$ be a transition of $\mathcal{D}_P$. There are two main cases that differ in the way of correspondence – x can be a method call event or a return event. For the sake of simplicity, assume that all the events specify all particular parameters and return values of method calls end returns. Note that if there is an event that does not specify parameter values or return value in P, it can be equivalently rewritten using alternative and all the possible values.

  - Let x be a method call event $m \uparrow (p_1, p_2, ...)$. Then the corresponding transition is $(\ell_P(s), true, l_x, \emptyset, \ell_P(t))$ where l_x is a label which stands for a call of the method m with appropriate parameters prescribed within the event. The label l_x takes the form $m(p_1, p_2, ...)$.
  - Let x be a method return event $m \downarrow : r$. Then the corresponding LTSA transition is $(\ell_P(s), g_x, \tau, \emptyset, \ell_P(t))$. The guard g_x is *true* if the method m does not return any value (i.e. its type is *void*). In the opposite case, i.e. if the method has a return value, the actual value is stored in the special purpose *Ret* variable as the method call has been issued in the previous transition. The guard $guard_x$ is then $Ret == \texttt{r}$.

  There are also special transitions for the s and f states

  - $(s, true, \tau, \emptyset, \ell_P(s_0))$ where s_0 is the initial state of $\mathcal{D}_P$;
  - $(\ell_P(p), true, \tau, \emptyset, f)$ for all p from the set of final states of states of $\mathcal{D}_P$.

On closer inspection, there is an obvious possibility of deadlock in the LTSA in case that a method returned an unexpected value. For example consider provision

$\mathtt{m}()$: $\mathtt{TRUE}$ for method $\mathtt{m}$ with no parameters and return type $\mathtt{Boolean} = \{\mathtt{TRUE}, \mathtt{FALSE}\}$. Figure 3.2 depicts the corresponding LTSA. There is only one transition after the call of the method m with the guard $Ret == \mathtt{TRUE}$ – the transition from q_2 to q_3. But there is no transition from q_2 for the case that m returns False, thus the execution would be blocked forever in this case.

Such a deadlock will be treated as a violation of the provision. It is however desirable to distinguish between this particular type of deadlock from the other deadlocks that may potentially occur (in other LTSA). It can be technically done by using transitions t with guard g_t pointing to the error state e from each state c immediately following a method call with a non-void return value. The guard g_t must be complementary to the guards on all the other transitions starting at c (let g_t be a negation of a disjunction of all the guards). Reachability of e then represents the deadlock. In the particular example there would be transition from q_2 to e with τ label and $\neg\, Ret == \mathtt{TRUE}$ guard.

Perceived strictly, the reachability of e is always a spurious error. It actually signals that the generated environment does not respect the provision. On the other hand, there might be a question whether it is even possible to create an environment that would call methods of the provisions correctly. For example take provision $(\mathtt{m}() : \mathtt{TRUE})\,|*$ and a reaction m that at least once returns *FALSE* and decides whether to return TRUE or *FALSE* in a way, which can not be affected from outside the incomplete model – for example using non-deterministic choice. In the case of non-deterministic choice it is surely impossible to create any valid environment and such model becomes unusable. In the case of deterministic choice there will exist a valid environment, but one might doubt whether such provision is a happy choice. Thus reachability of e can be a real error (the case that no valid environment exists) or a spurious error which could be taken as a recommendation to revisit the provision.

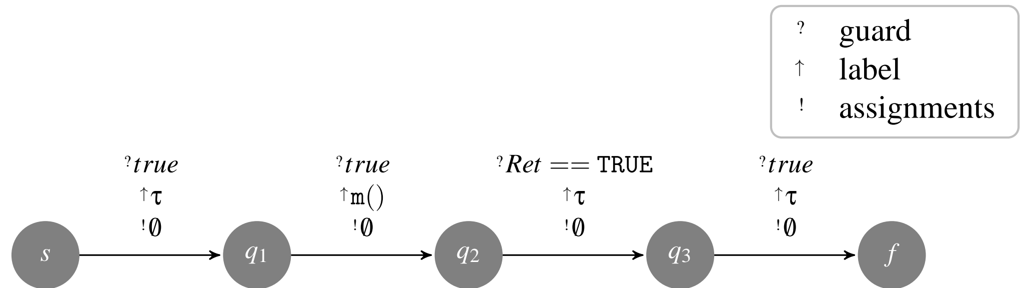

Figure 3.2: $LTSA_{\mathcal{D}_{\mathtt{m}():\mathtt{TRUE}}}$ – a LTSA derived from the provision expression $\mathtt{m}()$: $\mathtt{TRUE}$

Embedding $LTSA_{\mathcal{D}_P}$ in a thread and running several replicas of such threads in parallel will form a component, which generates some interleavings of words matching regular subexpressions of P. For example $LTSA_{\mathcal{D}_{\mathtt{m}()||\mathtt{n}()}}$ replicated twice would form words of $(\mathtt{m}()+\mathtt{n}())\,|\,(\mathtt{m}()+\mathtt{n}())$. Thus it is necessary to restrain the LTSA in order to generate words of the desired provision expression ($\mathtt{m}()||\mathtt{n}()$ in the example). In the

first place, the overall number of thread replicas is determined as the sum of upper parallel boundaries of all the entirely regular subexpressions of the provision expression. The number of thread replicas will be unlimited if the sum is equal to ω.

Furthermore, occurrences of the words matching individual regular subexpressions need to be limited with respect to the upper parallel boundaries. In the $\mathtt{m()}||\mathtt{n()}$ example, both $\mathtt{m()}$ and $\mathtt{n()}$ can be executed at most once (according to $\mathcal{U}_{\mathtt{m()}||\mathtt{n()}}(\mathtt{m()}) = \mathcal{U}_{\mathtt{m()}||\mathtt{n()}}(\mathtt{n()}) = 1$). It is then necessary to prevent both threads from executing the same part of the LTSA. Generally speaking, this can be achieved by using shared variables which count the number of the threads entering specific parts of the $LTSA_{\mathcal{D}_P}$. Transitions leading to such specific parts will increment the variables (counters) along their execution and special guards, that will be added to the transitions, will reflect the upper parallel boundaries.

Each state of the $LTSA_{\mathcal{D}_P}$ overlaps with some entirely regular subexpressions of P. The exact set of overlapped subexpressions is given by $\mathcal{C}_P \circ \ell_P^{-1}$ (as $\mathcal{C}_P\left(\ell_P^{-1}(q)\right)$ is a set of indexes of the subexpressions of P that the $\mathcal{D}_P$'s state $\ell_P^{-1}(q)$ overlaps with – if $\ell_P^{-1}(q)$ is defined for q). If a transition leaves a state which overlaps with more subexpressions than the target state, the transition will be called a *specializing* transition – for it enters a more specialized part of the $LTSA_{\mathcal{D}_P}$ with respect to the overlapped subexpressions. Let $J \subseteq I$ where I is an index set of entirely regular subexpressions of P and J is such that $\forall i \in J : \mathcal{U}_P(E_i) \neq \omega$, then c_J denotes a shared variable with domain dom_{c_J} of size $\sum_{i \in J} \mathcal{U}_P(E_i)$, $<_{\mathsf{c}_J}$ is a strict total order over dom_{c_J}, and 0_{c_J}, $\mathcal{M}_{\mathsf{c}_J}$ denote the minimal and the maximal element of dom_{c_J} respectively. The c_J variable acts as a counter of executions of all the specializing transitions that leave a state which overlaps with at least all the subexpressions indexed by J, and enter a state which overlaps with at least one subexpression with index in J.

More formally, let t be a specializing transition from p to q where neither p nor q is the final, the error nor the initial state. Then c_J acts as a counter for t if and only if $J \subset \mathcal{C}_P\left(\ell_P^{-1}(p)\right)$ and $\mathcal{C}_P\left(\ell_P^{-1}(q)\right) \subseteq J$. Moreover, let t be a transition from p to the final state f. Then c_J acts as a counter for t if and only if $J \subset \mathcal{C}_P\left(\ell_P^{-1}(p)\right)$ and $\mathcal{F}_P\left(\ell_P^{-1}(p)\right) \subseteq J$. The last statement captures the situation that the thread ends in a state, which overlaps with more expressions that it accepts. Then the thread's end in fact means its final specialization to the expressions which are accepted by the final state.

Let then t be a transition and $\{\mathsf{c}_{J_1}, \mathsf{c}_{J_2}, ..., \mathsf{c}_{J_k}\}$ a set of variables that act as counters of t. Let the transition t have a form (p, g_t, l_t, a_t, q) where p is the source state, g_t is a guard, l_t is a label, a_t is a set of assignments and q is the target state. Then let $t' = (p, g_{t'}, l_t, a_{t'}, q)$ be a transition derived from t which increments the counters and also guards their maximal value. Specifically

- $g_{t'} = g_t \wedge \left(\neg \mathtt{c}_{J_1} == \mathcal{M}_{\mathtt{c}_{J_1}}\right) \wedge \left(\neg \mathtt{c}_{J_2} == \mathcal{M}_{\mathtt{c}_{J_2}}\right) \wedge \ldots \wedge \left(\neg \mathtt{c}_{J_k} == \mathcal{M}_{\mathtt{c}_{J_k}}\right)$
- $a_{t'} = a_t \cup \{++\mathtt{c}_{J_1}, ++\mathtt{c}_{J_2}, \ldots ++\mathtt{c}_{J_2}\}$
  where $++\mathtt{c}_J$ stands for incrementation of the value of $\mathtt{c}_J$

Let $LTSA'_{\mathcal{D}_P}$ denote $LTSA_{\mathcal{D}_P}$ with every transition t replaced with t' appropriate to counters related to t. The $LTSA'_{\mathcal{D}_P}$ then respects the upper parallel boundaries of the subexpressions of P.

Note that the incrementation sign $++\mathtt{c}_J$ is not directly supported within the LTSA formalism. It stands for assigning an immediate successor (with respect to $<_{\mathtt{c}_J}$) of the actual value of the variable $\mathtt{c}_J$ back to $\mathtt{c}_J$. The case that actual value $\mathtt{c}_J$ does not have any successor, which is when the value is equal to $\mathcal{M}_{\mathtt{c}_J}$, is prevented by the guard in t'.

As already mentioned, $++\mathtt{c}_J$ is not supported by the LTSA formalism, however, it can be expressed using $|Dom_{\mathtt{c}_J}| - 1$ transitions – a transition for each value of $Dom_{\mathtt{c}_J}$ except $\mathcal{M}_{\mathtt{c}_J}$. The transition for a value v contains the guard $\mathtt{c}_J{==}v$ and the assignment $\mathtt{c}_J \leftarrow v'$ where v' denotes the immediate successor of v. Figure 3.3 shows an example of the expansion of an incrementation. Obviously, if there are more incrementation assignments on a transition, they can be gradually removed by using repetitive application of the scheme on all the newly formed transitions.

The last aspect influencing the proper cooperation of the threads formed by $LTSA'_{\mathcal{D}_P}$ lies in the incorporation of the lower parallel guards. In fact, a thread consisting of $LTSA'_{\mathcal{D}_P}$ always generates a word matching a regular subexpression of P. However, this is not always the desired behavior as or-parallel operator allows one of its arguments to be omitted. Thus it is necessary to allow threads to skip execution under some conditions. These conditions are given by the lower parallel guards.

The basic idea is to track whether a word matching individual regular subexpressions has been already generated by using shared variables equivalent to boolean variables. The variables would be initially set to `FALSE` and they would be assigned to `TRUE` on a transition targeting f from a state corresponding to appropriate final state of $\mathcal{D}_P$. Such variables would represent the propositional variables in the lower parallel guard for P. Finally there would be a new transition from s to f with a condition equivalent to $\mathcal{L}_P$ as the guard. This transition would allow a thread to skip its execution – let the transition be called *skipping* transition. Figure 3.4 presents the basic idea on an example.

Unfortunately, it is not necessary, that each predecessor of the state f corresponds to a state of $\mathcal{D}_P$ which contains only final states representing single regular subexpressions. Recall the function $\mathcal{F}_P$, the composition $\mathcal{F}_P \circ \ell_P^{-1}$ gives indexes of P's regular subexpressions that have a final state in the state corresponding to the argument – a state of $LTSA_{\mathcal{D}_P}$ and afterwards $LTSA'_{\mathcal{D}_P}$. In the example on Figure 3.4, there are both $|\mathcal{F}_P\left(\ell_P^{-1}(q_4)\right)| = 1$ and $|\mathcal{F}_P\left(\ell_P^{-1}(q_5)\right)| = 1$. But it is necessary to generalize the

$$
\begin{aligned}
Dom_{\mathtt{c}} &= \{\mathtt{ZERO}, \mathtt{ONE}, \mathtt{TWO}, \mathtt{THREE}\} \\
<_{\mathtt{c}} &\sim 0_{\mathtt{c}} = \mathtt{ZERO} <_{\mathtt{c}} \mathtt{ONE} <_{\mathtt{c}} \mathtt{TWO} <_{\mathtt{c}} \mathtt{THREE} = \mathcal{M}_{\mathtt{c}}
\end{aligned}
$$

?	guard
↑	label
!	assignments

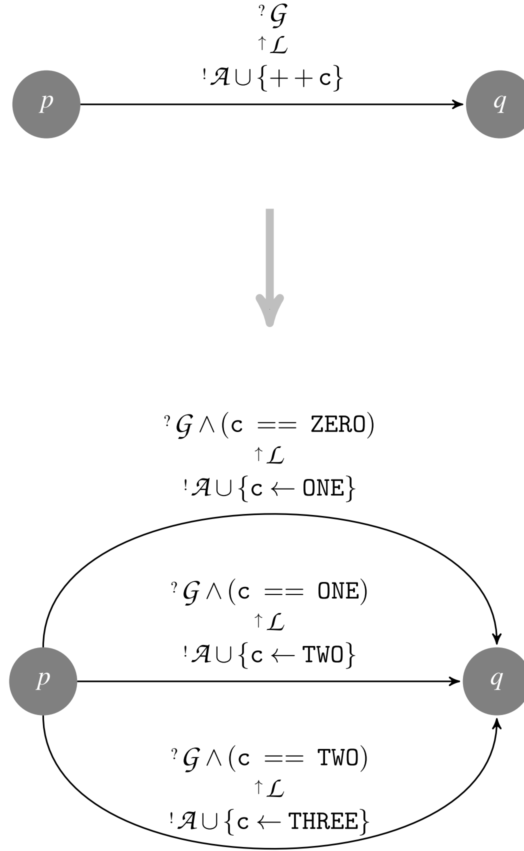

Figure 3.3: Expansion of an incrementation.

basic idea in order to support even such predecessor p of f, that $|\mathcal{F}_P\left(\ell_P^{-1}(p)\right)| > 1$.

Let p be a predecessor of f and let for p hold $\mathcal{F}_P\left(\ell_P^{-1}(p)\right) = \{n_1, n_2, ..., n_l\}$. When a thread takes the transition from p to f, it means that the sequence events generated by the thread matches all the subexpressions $E_{n_1}, E_{n_2}, ..., E_{n_l}$. Thus just one of $E_{n_1}, E_{n_2}, ..., E_{n_l}$ can be taken as fulfilled by the generated word. In other words, the thread *validates* just one of $\sigma_{E_{n_1}}, \sigma_{E_{n_2}}, ..., \sigma_{E_{n_l}}$ to be $true$. An interpretation of variables $\sigma_{E_1}, \sigma_{E_2}, ..., \sigma_{E_k}$ of the lower parallel guard is called *validated*, if for every i s.t. σ_{E_i} is $true$ there is at least one thread which has validated σ_{E_i} to be $true$. Finally the lower parallel guard formula is *validated*, if there is a validated interpretation of variables s.t. the formula is $true$ under the interpretation. Focusing back on the skipping transition, it is clear that it must be enabled just in case that the lower parallel guard $\mathcal{L}_P$ is validated.

In order to enable the skipping transition at the right moment, it is necessary to keep track of the finishing threads and formulate the guard of the skipping transition

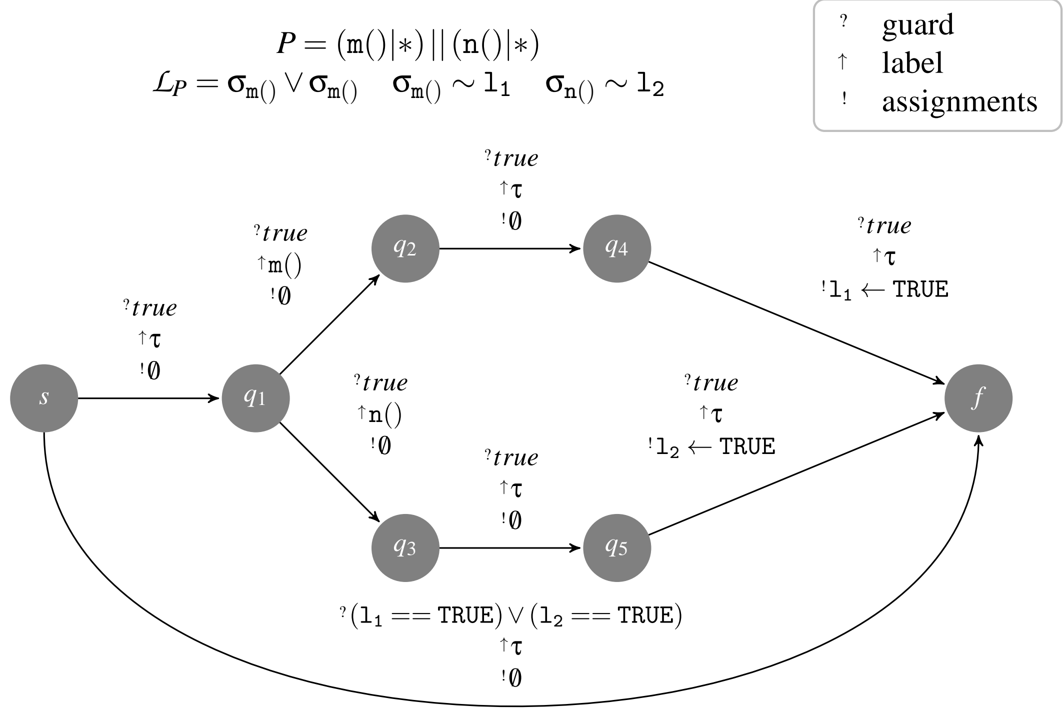

Figure 3.4: Example of application lower parallel guards.

in terms of $\mathcal{L}_P$ validation. The finishing threads can be tracked by counters specified as follows. Let p be a state of $LTSA'_{\mathcal{D}_P}$ such that there is a transition t from p to the final state f. Let $\mathcal{F}_P\left(\ell_P^{-1}(p)\right) = \{n_1, n_2, ..., n_\kappa\} =: F$. Then there is a shared variable $\mathtt{l}_F$ with domain of size $|F|+1$ and a strict total order $<_F$ over the domain. The initial value of $\mathtt{l}_F$ is the minimal element of its domain 0_F. Additionally, let $\mathcal{M}_F$ denote the maximal element of the domain. The transition $t = (p, g_t, l_t, a_t, f)$ will be replaced by two parallel transitions t'' and t''':

- $t'' = (p, g_t \wedge (\mathtt{l}_F == \mathcal{M}_F), l_t, a_t, f)$
- $t''' = (p, g_t \wedge (\neg \mathtt{l}_F == \mathcal{M}_F), l_t, a_t \cup \{++\mathtt{l}_F\}, f)$

Let $LTSA''_{\mathcal{D}_P}$ be an LTSA formed by replacing all the transitions targeting the final state f in $LTSA'_{\mathcal{D}_P}$. The $LTSA''_{\mathcal{D}_P}$ then tracks its last transition within the $\mathtt{l}_{\{...\}}$ variables.

Now it is needed to express the lower parallel guard $\mathcal{L}_P$ using the $\mathtt{l}_{\{...\}}$ variables for the guard condition on the skipping transition. A possible way is to find a formula $\mathcal{L}'_P$ in the disjunctive normal form containing no negations which is equivalent to $\mathcal{L}_P$. Such formula clearly exists since $\mathcal{L}_P$ contains no negations. However, it can be exponentially larger than $\mathcal{L}_P$. Let $\mathcal{L}'_P$ be a disjunction of conjunctions $C_1, C_2, ..., C_n$. Then for every C_i a formula L_i consisting of $\mathtt{l}_{\{...\}}$ variables equivalent to C_i in terms of validation may be constructed. Finally the disjunction $L_1 \vee L_2 \vee ... \vee L_n$ is equivalent to $\mathcal{L}_P$ (in terms of validation).

In the following, a possible way in which the L_i formula can be created is proposed. Let there are exactly $\mathtt{l}_{F_1}, \mathtt{l}_{F_1}, ..\mathtt{l}_{F_m}$ counters tracking all the transitions to the final state f in $LTSA''_{\mathcal{D}_P}$. The sets $F_1, F_2, ..., F_m$ are mutually different subsets of I, the index set of the maximal entirely regular subexpressions of P. Recall that the domain of $\mathtt{l}_{F_i}$ has $|F_i|+1$ elements and there is the strict total order $<_{F_i}$ over the domain. Let b_{F_i} be a bijection $b_{F_i} : Dom_{\mathtt{l}_{F_i}} \rightarrow \{0, 1, ..., |F_i|\}$ which respects the order [1]. Then let $J = \{j_1, j_2, ..., j_n\}$ be a subset of I such that

$$C_i = \bigwedge_{j \in J} \sigma_{E_j}$$

Let $\mathbf{F}_j$ denote the set $\{F_i \mid 1 \leq i \leq m \wedge j \in F_i\}$ and $\mathbf{F}_J$ denote $\mathbf{F}_{j_1} \times \mathbf{F}_{j_2} \times ... \times \mathbf{F}_{j_n}$. Then the formula L_i that is equivalent to C_i in terms of validations has the form

$$L_i = \bigvee_{\eta \in \mathbf{F}_J} \left(\bigwedge_{j=1}^{n} \mathtt{l}_{F_j} \geq b_{F_j}^{-1} \left(min \left(\theta_j^{\eta}, |F_j| \right) \right) \right) \tag{3.12}$$

where θ_j^{η} is the number of occurrences of F_j in η, more precisely, the number of different i s.t. $\eta_{(i)} = F_j$.

Let L_P be the disjunction of $L_1, L_2, ..., L_n$. Then $LTSA_P$ denotes an LTSA created from $LTSA''_{\mathcal{D}_P}$ by adding a skipping transition t in the form $(s, L_P, \tau, \emptyset, f)$ where s, f are the initial and final states of $LTSA''_{\mathcal{D}_P}$ respectively. Let $\mathcal{C}$ be a component formed by the thread consisting in $LTSA_P$. Let the thread be replicated as many times, as the sum of upper parallel boundaries of all entirely regular subexpressions of P (in case that the sum is ω, the number of threads will be unlimited). In other words, let the sum form a *replication factor* of the thread. It is worth mentioning that the component also contains all the shared variables $\mathtt{c}_{...}$ and $\mathtt{l}_{...}$ that appear on the transitions of $LTSA_P$. The component $\mathcal{C}$ then preforms all the possible traces of events accepted by P within all the possible execution paths of $LTSA_P$ while the reactions respond correctly to the events with respect to P. On the other hand, if a reactions responds incorrectly – i.e. it yields an unexpected return value, it is immediately apparent as an error or more precisely as reachability of the error state e.

Note that $\geq$ operator is not defined within LTSA guards. On the other hand, having a strict total order $<_v$ over Dom_v, the domain of variable v, a set $Dom_v^{\geq w} \subseteq Dom_v$ containing exactly the elements that are not less than w is well defined for any $w \in Dom_v$. Formally, $Dom_v^{\geq w} = \{x \mid x \in Dom_v \wedge \neg(x <_v w)\}$ for any $w \in Dom_v$. Then any expression in from $v \geq w$ such that $w \in Dom_v$ is equivalent to $v \in Dom_v^{\geq w}$ and that

[1] $\forall x, y \in Dom_{\mathtt{l}_{N_i}} : b_{N_i}(x) < b_{N_i}(y) \iff x <_{N_i} y$

can be scripted in form of a guard expression as

$$\bigvee_{X \in Dom_v^{\geq W}} (v{==}X)$$

For example, let P be the provision expression $((\mathtt{m}()+\mathtt{n}()) \mid\mid \mathtt{p}()) \mid \mathtt{m}()$. Let E_1, E_2, E_3 be the entirely regular subexpressions of P as they appear in P, i.e. $E_1 = \mathtt{m}()+\mathtt{n}()$, $E_2 = \mathtt{p}()$ and $E_3 = \mathtt{m}()$. Then there will be three predecessors of the final state of $LTSA_P$ – q_5, q_6 and q_7 on Figure 3.5. The first corresponds to the words matching E_1, second corresponds to the words matching E_1 intersected with the words matching E_3, and the last represents E_2. Hence there are counters $\mathtt{l}_{\{1,3\}}$, $\mathtt{l}_{\{1\}}$ and $\mathtt{l}_{\{2\}}$ for the lower parallel guard and the lower parallel guard itself is $\mathcal{L}_P = (\sigma_{E_1} \vee \sigma_{E_2}) \wedge \sigma_{E_3}$. The disjunctive normal form $\mathcal{L}'_P$ is then $(\sigma_{E_1} \wedge \sigma_{E_3}) \vee (\sigma_{E_2} \wedge \sigma_{E_3})$. The domains of the counters along with the order are

$$\begin{aligned} Dom_{\mathtt{l}_{\{1\}}} = Dom_{\mathtt{l}_{\{2\}}} &= \{\mathtt{V0},\mathtt{V1}\} \qquad \mathtt{V0} < \mathtt{V1} \\ Dom_{\mathtt{l}_{\{1,3\}}} &= \{\mathtt{V0},\mathtt{V1},\mathtt{V2}\} \quad \mathtt{V0} < \mathtt{V1} < \mathtt{V3} \end{aligned}$$

From 3.12, the representation of $\mathcal{L}_P$ using the counters is

$$\begin{aligned} L_P = &\left(\mathtt{l}_{\{1\}} \geq \mathtt{V1} \wedge \mathtt{l}_{\{1,3\}} \geq \mathtt{V1}\right) \vee \left(\mathtt{l}_{\{1,3\}} \geq \mathtt{V2}\right) \\ &\vee \left(\mathtt{l}_{\{1\}} \geq \mathtt{V1} \wedge \mathtt{l}_{\{2\}} \geq \mathtt{V1}\right) \vee \left(\mathtt{l}_{\{1,3\}} \geq \mathtt{V1} \wedge \mathtt{l}_{\{2\}} \geq \mathtt{V1}\right) \end{aligned}$$

Figure 3.5 shows the resulting $LTSA_P$ for the example.

3.2.2 Imperative Parts

This section describes the way the imperative parts of a TBP model form a VASS which can be later used for testing reachability of particular thread states by the coverability of the appropriate VASS configurations. The imperative parts consist of the threads and the reactions represented by the LTSA structures. For a straightforward representation using VASS the reactions need to substitute for the calling transitions. This substitution is very similar to the concept of *inline* functions supported by some programming languages such as *C*, thus the term *inlining* or *inline expansion* will be used in this text. After all reactions are inlined into their callers, the imperative parts are formed by a set of threads only. There are no actual calls (but the labels are retained) and no need of stacks which is the main benefit for the straightforward representation using VASS.

Inline Expansion

The expansion consists in replacing the LTSA transition which calls a method with a LTSA fragment derived from the appropriate reaction. A call transition is a transi-

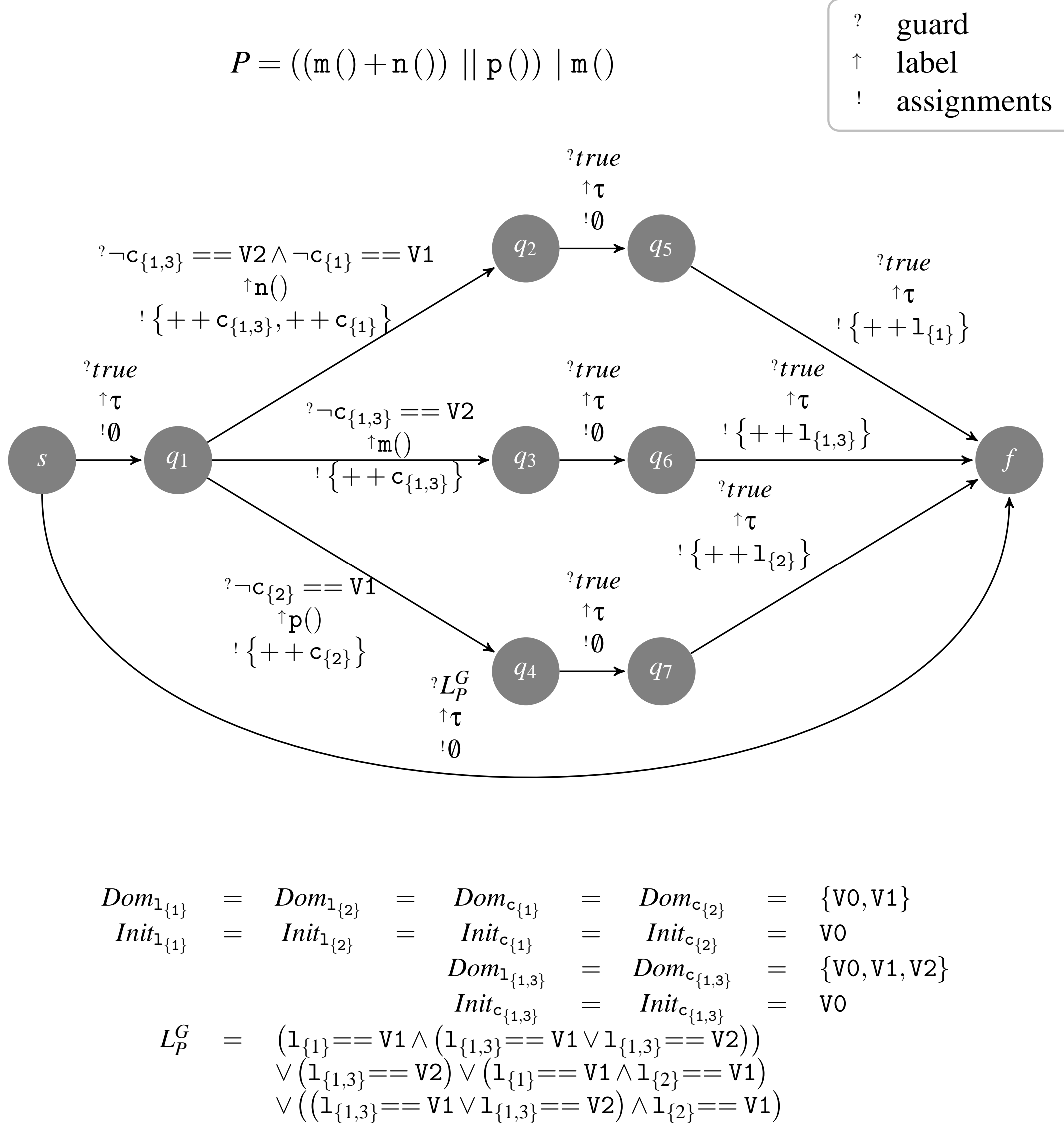

Figure 3.5: Example of $LTSA_P$.

tion in the form (p, g_c, l_c, a_c, q) where the label l_c is a parametrized label in the form $m(p_1, p_2, ..., p_k)$. Generally, a call transition is the only one transition entering its target state and may be followed by any number of transitions which refer *Ret* variable either within its guard or assignments. Recall that *Ret* variable is the special purpose variable for storing return value of a reaction.

The LTSA representing a reaction is generally arbitrary LTSA structure such that there is no transition starting at any final state – the final states represent states after `return`, thus any continuation is useless. Without loss of generality, only one final state can be assumed as all the transitions pointing to the original final states can be redirected into a single final state. The transitions pointing to the final state may contain an assignment to *Ret*, depending on whether the reaction represents a method with a return value. Note that the successors of a call transition can refer *Ret* only if the reaction returns a value (assigns *Ret*).

The LTSA fragment $\mathcal{F}_{c,R}$ which replaces a call transition $c = (p, g_c, l_c, a_c, q)$ in a call site with local variables L_c (i.e. either a thread or another reaction) of a reaction R with local variables L_r and a parameter mapping function $\rho : \mathbb{N} \to L_r$ is formed using following steps.

1. Rename the local variables L_r.
   The variables L_r need to be renamed in order not to clash with variables of the call site L_c. Thus let L'_r be an alternative set of the local variables such that $L'_r \cap L_c = \emptyset$ and let $r : L_r \to L'_r$ be a bijection that captures the renaming. All occurrences of each variable $l \in L_r$ in the LTSA representing the behaviour of the reaction need to be replaced with $r(l)$.

2. Assign parameter values.
   In order to assign parameters values, it is necessary to add a new initial state s to the LTSA representing the behaviour of the reaction, which will precede the original initial state s_0. The parameter values will be assigned to appropriate local variables within the transition $(s, g_c, \tau, a_c \cup \xi, s_0)$ where ξ is the set of assignments $\{r(\rho(i)) \leftarrow p_i \mid 1 \leq i \leq n\}$; n is the number of parameters and p_i is the value of the i-th parameter as stated in l_c. Note that the transition reuses guard and assignments from the original call site.

3. Mark entry and end transitions with labels.
   In order to keep track of called methods, labels corresponding to the appropriate events need to be added. Let $l'_c = m \uparrow (p_1, p_2, ..., p_n)$ be a parametrized label which represents the call event appropriate to l_c and $l'_r = m \downarrow : Ret$ or $l'_r = m \downarrow$ represent appropriate return event (depending on whether the reaction returns a value). Then the label l'_c must be added to the entry transition (i.e. the transition from s to s_0 from the previous step) and l'_r to the end transitions (i.e. the transitions pointing the only final state).

The replacement itself is fairly straightforward. The entry transition of $\mathcal{F}_{c,R}$, which is the transition from its initial state s, will be moved so that it will start at p, the source state of the call transition c. The end transitions, which are the transitions targeting the only final state of $\mathcal{F}_{c,R}$, will be redirected to q, the state which c points to. Then c will be removed and the set of local variables associated with the call site will be $L_c \cup L'_r$.

Moreover, it is a great opportunity to get a rid of the *Ret* variable within the inlining. The *Ret* variable has some peculiar properties (e.g. it is not properly typed) that would eventually complicate its representation within VASS. The *Ret* variable will be replaced with a regular local variable of the same type as the reaction's return type. Let $\mathtt{r}_{tmp}$ denote such a variable. However, the actual name of $\mathtt{r}_{tmp}$ must be chosen carefully in order not to clash with any variable in L_c or L'_r. Then $\mathtt{r}_{tmp}$ will be added

to the set of local variables associated with the call site, the set will then be equal to $L_c \cup L'_r \cup \left\{ \mathtt{r}_{tmp} \right\}$. Finally, let all the occurrences of *Ret* within all the transitions pointing to q or starting at q (i.e. assignments to *Ret* or *reads* of *Ret* respectively) be replaced with $\mathtt{r}_{tmp}$.

Threads to VASS

After all the method calls are expanded, the imperative parts of a TBP model are represented by threads $T_1, T_2, ...T_n$, where T_i takes the form (L_i, Γ_i). Having G a set of global variables, L_i is a set of local variables used within the thread and Γ_i is an LTSA representation of the thread's behavior, which refers variables $G \cup L_i$. Moreover, there is a replication function $\mathcal{R} : \{1, 2, ..., n\} \to \mathbb{N} \cup \{\omega\}$, which assigns a replication factor to each thread. A thread will be instantiated as many times, as its replication factor – ω denotes unlimited number of thread instances. Recall that the replication factor is equal to the sum of upper parallel boundaries for the threads created as environment from an unbound provision and 1 for regular threads derived from TBP specification.

In order to be able to simulate the threads using VASS, it is necessary to provide an unique identifier for individual LTSA states, or in other words, some sort of program counter is needed. Let θ_i be a set of all states of Γ_i, the LTSA for i-th thread, and Θ be a set of all states of all the threads, $\Theta = \theta_1 \cup \theta_2 \cup ... \cup \theta_n$. Then let $\phi : \Theta \to \{1, 2, ..., |\Theta|\}$ be a bijection which maps state to its integral identifier.

To keep relation between thread state and the LTSA state, it is necessary to materialize ϕ within the LTSA structures as a program counter variable. The program counter is specific to each thread, thus it will be retained within a local variable. Let D be a set of constants of size $|\Theta|$ which will act as a domain of variables representing the program counters and let $d : D \to \{1, 2, ..., |\Theta|\}$ be a bijection evaluating D. Then for all $i = 1, 2, ...n$ let $pc_i \notin L_i$ be a variable with domain $Dom_{pc_i} = D$ and initial value $Init_{pc_i} = d^{-1}(\phi(s_i))$, where s_i is the initial state of Γ_i. The variable pc_i will act as program counter for thread T_i, thus let $T_i^{pc} = \left(L_i^{pc_i}, \Gamma_i^{pc}\right)$ be the thread equipped with program counter corresponding to T_i, where $L_i^{pc} = L_i \cup \{pc_i\}$ and Γ_i^{pc} is a LTSA s.t.

- the set of states, the initial state, the final states and the labels of Γ_i^{pc} are the same as in Γ_i;
- transitions of Γ_i^{pc} are in one-to-one correspondence with those of Γ_i;
- if $t = (p, g_t, l_t, a_t, q)$ is a transition of Γ_i and t^{pc} is the corresponding transition of Γ_i^{pc}, then $t^{pc} = \left(p, g_t, l_t, a_t \cup \left\{pc_i \leftarrow d^{-1}(\phi(q))\right\}, q\right)$.

Simply put, each transition assigns the identifier of its target state to the program counter.

For explicit encoding of the threads of TBP model into VASS, all local states need to be enumerated. The set $\mathcal{L}$ of all local states is a union of sets of local states of all the threads. The local states $\mathcal{L}_{T_i^{pc}}$ of a thread T_i^{pc} are given by all possible valuations of local variables L_i^{pc}. Let ζ be the number of all local states, i.e. $|\mathcal{L}|$, then the set $\mathcal{L}$ can be indexed with $\{1,2,...,\zeta\}$ so that $\mathcal{L} = \{\ell_1, \ell_2, ..., \ell_\zeta\}$. Conversely, it is not necessary to explicitly assign numbers to the global states. The global states are formed by all possible valuations of global variables G, let the set of all global states be denoted by $\mathcal{G}$.

Let $s \in \mathcal{G}$ be a global state, then let the valuation function γ_G that maps all global variables to the values corresponding to s be denoted a γ_s. Analogously, let $\ell \in \mathcal{L}_{T_j^{pc}}$ be a local state of a thread $T_j^{pc} = \left(L_j^{pc}, \Gamma_j^{pc}\right)$, then let the valuation function $\gamma_{L_j^{pc}}$ that maps all local variables in L_j^{pc} to the values corresponding to ℓ be denoted as γ_ℓ. Additionally, let ψ be a function mapping local states to the states of LTSA. More precisely, let ℓ be a local state s.t. $\gamma_\ell(pc_i)$ is defined, then $\psi(\ell)$ is $\Phi^{-1}(d(\gamma_\ell(pc_i)))$, which is the LTSA state appropriate to ℓ. ψ is well defined because for every $\ell \in \mathcal{L}$ there is exactly one $i \in \{1,2,...,n\}$ such that $\gamma_\ell(pc_i)$ is defined.

Let $\mathcal{W} = (\mathcal{S}, c_0, \delta)$ be a ζ-dimensional VASS such that

- $\mathcal{S} = \mathcal{G}$, i.e. the states are formed by the set of all valuations of global variables
- $c_0 = (s, v)$ where s evaluates the global variables as their initial values and $v \in (\mathbb{N}_0)^\zeta$ s.t.

  $$v_{(i)} = \begin{cases} \mathcal{R}(j) & \text{if } \exists j \in \{1,2,...,n\} : \mathcal{R}(j) \neq \omega \wedge \forall v \in L_j^{pc} : \gamma_{\ell_i}(v) = Init_v \\ 0 & \text{otherwise} \end{cases}$$

  i.e. $v_{(i)}$ is equal to $\mathcal{R}(j) \neq \omega$ if i-th local state evaluates the local variables of thread T_j^{pc} as their initial values.

- $(s_1, s_2, w) \in \delta$ if and only if

  (i) w contains just two non-zero coordinates $w_{(i)} = -1$ and $w_{(j)} = 1$ and there is a LTSA transition $t = (\psi(\ell_i), g_t, l_t, a_t, \psi(\ell_j))$ such that

  - g_t evaluates to $true$ under γ_{s_1} and γ_{ℓ_i}
  - $a_t \subseteq \{g \leftarrow \gamma_{s_2}(g) \mid g \in G\} \cup \{l \leftarrow \gamma_{\ell_j}(l) \mid l \in L_1^{pc} \cup L_2^{pc} ... \cup L_n^{pc}\}$
  - $g \leftarrow \gamma_{s_2}(g) \in a_t$ for every global variable g such that $\gamma_{s_1}(g) \neq \gamma_{s_2}(g)$
  - $l \leftarrow \gamma_{\ell_j}(l) \in a_t$ for every local variable l such that $\gamma_{\ell_i}(l) \neq \gamma_{\ell_j}(l)$

  or

  (ii) w contains just one non-zero coordinate $w_{(i)} = 1$, $s_1 = s_2$ and there is j such that ℓ_i evaluates exactly all the local variables $L_{T_j^{pc}}$ of j-th thread T_j^{pc}

as their initial values and the replication factor of the j-th thread $\mathcal{R}(j)$ is ω.

Note that the transitions (i) specify regular transitions derived from the threads' behavior, whereas the transitions (ii) represent creation of threads with unlimited replication factor.

The stack-based computation of a model containing the threads $T_1^{pc}, T_2^{pc}, \ldots, T_n^{pc}$ where i-th thread is instantiated $\mathcal{R}(i)$-times corresponds to a simulation of the VASS $\mathcal{W}$ described above. Recall that the stack-based computation state consists of valuation of the global variables γ_s and a set of stacks for the threads. Each stack is an ordered sequence of records in the form (l, γ_l) where l is a state of LTSA and γ_l is a valuation of local variables. Note that for this case, as all the method calls has been inlined, each stack will always contain at most one record. Let $S_{i,j}$ denote the stack corresponding to execution of j-th instance of the i-th thread for $1 \leq i \leq n$ and $1 \leq j \leq \mathcal{R}(i)$. Also note, that this extension of the stack-based computation for the unlimited number of thread instances induces that the set of stacks is unbounded. Additionally let the stacks corresponding to the threads with unlimited replication factor be initially empty.

Such a computation state $\mathbf{S}$ corresponds to a configuration $c = (s, v)$ if the global state s corresponds to γ_s and all the coordinates of the vector v reflect the stacks of $\mathbf{S}$. Let $k \in \mathbb{N}$ s.t. $1 \leq k \leq \zeta$, then k-th coordinate of v represents the number of threads which reside in the local state ℓ_k. The state corresponds to a valuation γ_k which also contains the information about the current LTSA state $l_k = \psi(\ell_k)$. Then the k-th coordinate reflects the stacks in the computation state if there is exactly $v_{(k)}$ stacks with the top-most element equal to (l_k, γ_k).

The transitions that can be executed from a computation state $\mathbf{S}$ correspond to the transitions available from c in $\mathcal{W}$. Recall δ in the definition $\mathcal{W}$, concretely the part (i); it uses exactly the transitions that can be taken from $\mathbf{S}$. Additionally, the part (ii) defines transitions corresponding to instantiation of a new thread instance – just for threads with unbounded replication factor, if there are any. Note that for the threads with unbounded replication there is an unlimited number of stacks. As these stacks were initially empty, there is always at least one empty stack which may be used for the newly created thread. Let l_i^0 denote the initial state of the LTSA of the i-th thread, then instantiation of i-th thread corresponds to adding record $(l_i^0, \gamma_{L_i}^0)$ onto stack $S_{i,j}$ where j is the smallest number s.t. $S_{i,j}$ is empty. With this obvious extension of the stack-based computation, the VASS transitions available from c correspond to the computation transitions available from $\mathbf{S}$.

Clearly, the initial state of the stack-based computation $\mathbf{S}_0$ corresponds to the initial configuration $c_0 = (s, v)$ as the stacks of the threads with unlimited replication factor are empty in $\mathbf{S}_0$.

However, the explicit VASS representation brings huge overhead induced by enumerating of all possible local states in practice. This overhead is eliminated within the symbolic version of counter abstraction, which does not require existence of VASS at all. Recall that the basic idea is that the model checking phase – i.e. construction of the Karp and Miller tree – is interleaved with context-aware exploration of the program – i.e. LTSAs of the threads in our case. The VASS is then not necessary at all, though, in metaphorical sense, portions of the concrete VASS are constructed on-the-fly within the exploration.

Algorithm 1: Karp-Miller Tree Construction (adapted and adjusted from [Zha09])

```
input : ℙ a program or a model , Initial a set of initial configurations
begin
    Unexplored ← Reached ← Initial
    Pred ← ∅
    while Unexplored ≠ ∅ do
        remove c = (s, L, f : L → ℕ ∪ {ω}) from Unexplored
        P ← PredecessorsOf (c, Pred)        // all predecessors of c till the root
        if c ∉ P then
            for l ∈ L do
                Steps ← Explore (ℙ, c, l)
                for (s', L_abandoned, L_taken) ∈ Steps do
                    L' ← (L \ L_abandoned) ∪ L_taken
                    f'(x) = { f(x) − 1   x ∈ L_abandoned
                            { f(x) + 1   x ∈ L_taken ∧ x ∈ L
                            { 1          x ∈ L_taken ∧ x ∉ L
                            { f(x)       otherwise
                    c' ← Omega (s', L', f', P ∪ {c})   // introduce ω to the counters
                    insert c' into Unexplored
                    add c' to Reached
                    add (c, c') to Pred                // c is a predecessor of c'
```

Algorithm 1 presents a version of the Karp-Miller tree construction, which abstracts from the actual representation of the model. The algorithm works with a slightly different representation of the configurations; a configuration is newly represented by a tuple $(s, L, f : L \to \mathbb{N} \cup \{\omega\})$ where s is a global state, i.e. a valuation of global variables, L is a set of occupied local states and f maps the local states to the number of thread instances that reside in it. The f function captures only the local states, that are occupied in the configurations. Note that if there is an enumeration of all local states, this representation of the configuration can be totally mapped to the explicit one, because f can be obviously extended to all local states: $f(l) = 0$ for $l \notin L$.

Algorithm 2: The OMEGA Procedure (adapted from [Zha09])

```
function OMEGA (s, L, f, Predecessors)
    for (s_p, L_p, f_p) ∈ Predecessors do
        if COVERS ((s, L, f), (s_p, L_p, f_p)) then
            for l ∈ L do
                f'(x) = { ω      l ∉ L_p
                        { ω      l ∈ L_p ∧ f(l) > f_p(l)
                        { f(l)   otherwise
    return (s, L, f')

function COVERS ((s, L, f), (s', L', f'))        // the former covers the later
    if ¬(L' ⊆ L) then
        return false
    for l' ∈ L' do
        if f'(l') > f(l') then
            return false
    return true
```

The EXPLORE function in Algorithm 1 performs the context-aware exploration of the input model $\mathbb{P}$. The context is specified by c, the current configuration, and the exploration is limited to steps involving local state l. The function yields viable steps represented in the from of triplets (s, L_a, L_t), where s is a new global state, L_a is a set of local states that have been abandoned within the step, and L_t is a set of local states that have been taken within the step.

For illustration, this paragraph presents the way the EXPLORE function works for a k-dimensional VASS $(\mathcal{S}, c_0, \delta)$. It takes all transitions starting at the global state of the current context. Then it filters out transitions that do not contain negative value on the coordinate corresponding to given local state which limits the exploration. Moreover it filters out all transitions that contain negative value on the coordinate corresponding to a local state which is not occupied in the context (or there is not enough threads). Then the transitions are transformed to the resulting triplets – s is the target state, L_a contains all local states those coordinates are negative as many times as the absolute value of the coordinate, L_t contains local state those coordinates are positive as many times as the value of the coordinate. Note that in general L_a and L_t are multi-sets. On the other hand, simple sets are enough in case of VASS derived from TBP model since $\delta \subseteq \left(\mathcal{S} \times \mathcal{S} \times \{-1, 0, 1\}^k\right)$. More precisely, let the set of all local states be enumerated, i.e. $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_k\}$, and let $\delta \subseteq \left(\mathcal{S} \times \mathcal{S} \times \{-1, 0, 1\}^k\right)$. Then

$$\left(s',L_a,L_t\right) \in \textsc{Explore}\left(\left(\mathcal{S},c_0,\delta\right),\ \left(s,L,f:L\to\mathbb{N}\cup\{\omega\}\right),\ l\right)$$
$$\Updownarrow$$
$$\left(\exists\left(s,s',v\right)\in\delta\right)\left(\forall i\in\{1,2,...,k\}\right)\ \text{s.t.}$$
$$\left(v_{(i)}+f\left(\ell_i\right)\geq 0\right)\wedge\left(\ell_i\in L_a \iff v_{(i)}=-1\right)\wedge\left(\ell_i\in L_t \iff v_{(i)}=1\right)$$

Moreover, it is convenient to completely omit the VASS and let EXPLORE operate on the LTSA representation of the threads directly. For this purpose, let $\mathbb{P}$ be a triplet $(\mathcal{G},T,\mathcal{R})$ where $\mathcal{G}$ is a set of all possible valuations of global variables G, T is a set of threads $T_1,T_2,...,T_n$ in the form $T_i=\left(L_i^{pc},\Gamma_i^{pc}\right)$, and $\mathcal{R}:\{1,2,...,n\}\to\mathbb{N}\cup\{\omega\}$ is the replication function. The EXPLORE function must then return the steps corresponding to viable LTSA transitions from given context as well as the steps representing creation of new thread instances for threads with unlimited replication factor. Since these two aspects are disjoint and the later does not depend on the local state l, EXPLORE can be split as follows

$$\textsc{Explore}\left(\mathbb{P},\ c,\ l\right):=\textsc{Explore}'\left(\mathbb{P},\ c,\ l\right)\ \cup\ \textsc{NewThreads}\left(\mathbb{P},\ c\right)$$

NEWTHREADS $(\mathbb{P},\ (s,L,f))$ contains triplets (s',L_a,L_t) such that $s'=s$, $L_a=\emptyset$ and L_t contain one initial local state for each thread with unlimited replication factor. In particular, a triplet $(s,\emptyset,\{l\})\in$ NEWTHREADS $(\mathbb{P},\ (s,L,f))$ if and only if there is such $i\in\{1,2,...,n\}$ that $\psi(l)$ is the initial state of Γ_i^{pc}, the replication factor $\mathcal{R}(i)$ is equal to ω and $\gamma_l(v)=Init_v$ for all local variables $v\in L_i^{pc}$.

EXPLORE$'$ $(\mathbb{P},\ (s,L,f),\ l)$ contains a triplet (s',L_a,L_t) if and only if $L_a=\{l\}$, there is l' s.t. $L_t=\{l'\}$, there is $i\in\{1,2,...,n\}$ s.t. $\psi(l)$ is a state of Γ_i^{pc} and there is a transition $t=(\psi(l),g_t,l_t,a_t,\psi(l'))$ in the transition relation δ of Γ_i^{pc} s.t.

- g_t evaluates to *true* under γ_s and γ_l
- $a_t\subseteq\{v\leftarrow\gamma_{s'}(v)\mid v\in G\}\cup\left\{v\leftarrow\gamma_{l'}(v)\mid v\in L_i^{pc}\right\}$
- $v\leftarrow\gamma_{s'}(v)\in a_t$ for every global variable v such that $\gamma_s(v)\neq\gamma_{s'}(v)$
- $v\leftarrow\gamma_{l'}(v)\in a_t$ for every local variable v such that $\gamma_l(v)\neq\gamma_{l'}(v)$

Algorithm 3 presents the EXPLORE$'$ function.

The EXPLORE presented so far works well with the initial configuration $c_0=(s,L,f:L\to\mathbb{N}\cup\{\omega\})$ where $s\in\mathcal{G}$ s.t. γ_s evaluates all the global variables as their initial values, $l\in L$ if and only if $\exists i\in\{1,2,...,n\}$ s.t. $\psi(l)$ is an initial state of Γ_i^{pc} and γ_l evaluates all the local variables in L_i^{pc} as their initial values and $\mathcal{R}(i)\neq\omega$. For f holds $f(l)=\mathcal{R}(i)$ where i is a number s.t. $\psi(l)$ is a state of Γ_i^{pc} for all $l\in L$. Note that this definition is completely analogous to the construction of VASS $\mathcal{W}$ from the threads.

On the other hand, the symbolic model allows the infinite number of thread instances to occupy thread-local states even in the initial configuration. Therefore, there would be no need to manage transitions for creating new threads in each step if the initial configuration already reflected threads with unlimited replication. Hence, an alternative, and more elegant, approach is to define initial configuration $c_0 = (s, L, f)$ as follows:

- s is the initial global state, i.e. $s \in \mathcal{G}$ s.t. $\forall v \in G : \gamma_s(v) = Init_v$
- $l \in L$ if and only if l is an initial local state of a thread, i.e. $\exists i \in \{1, 2, ..., n\}$ s.t. $\psi(l)$ is the initial state of Γ_i^{pc} and $\forall v \in L_i^{pc} : \gamma_l(v) = Init_v$
- $f(l) = \mathcal{R}(i)$ where i is a number s.t. $\psi(l)$ is a state of Γ_i^{pc} ($\forall l \in L$).

The result of the exploration phase no longer contains the creation of new threads, thus let the exploration phase be represented only by the function EXPLORE′.

Algorithm 3: The EXPLORE′ Procedure

function EXPLORE′ $((\mathcal{G}, T, \mathcal{R}),\ (s, L, f),\ l)$
  $Result \leftarrow \emptyset$
  let $\left(L_i^{pc}, \Gamma_i^{pc}\right) \in T$ *s.t.* $\psi(l) \in \Gamma_i^{pc}$ **begin**   // pick the thread which l belongs to
    for $(p_t, g_t, l_t, a_t, q_t) \in \delta_{\Gamma_i^{pc}}$ *s.t.* $p_t = \psi(l)$ **do**   // all transitions from $\psi(l)$
      if VALUEOF $(g_t,\ s,\ l) = true$ **then**   // is the guard is fulfilled?
        $(s', l') \leftarrow$ ASSIGN $((s, l),\ a_t)$
        add $(s', \{l\}, \{l'\})$ to $Result$
  return $Result$

function ASSIGN $((s, l),\ Assignments)$

$$\gamma_{glob}(x) = \begin{cases} c & x \leftarrow c \in Assignments \\ \gamma_s(x) & \text{otherwise} \end{cases} \qquad x \in dom(\gamma_s)$$

$$\gamma_{loc}(x) = \begin{cases} c & x \leftarrow c \in Assignments \\ \gamma_l(x) & \text{otherwise} \end{cases} \qquad x \in dom(\gamma_l)$$

  $s' \leftarrow$ the state corresponding to γ_{glob}
  $l' \leftarrow$ the state corresponding to γ_{loc}
  return (s', l')

3.2.3 Bound Provisions

For the verification of correctness of a TBP model, the provisions are an all-important part of the protocol. In essence, they determine the desired properties of the imperative

part of the model. This section will describe the way in which the bound provisions are captured within checking an incomplete TBP model using symbolic counter abstraction.

Recall that in the incomplete model the unbound provisions are restricted so that each forms a simultaneously regular expression or does not contain full reentrancy. The reason is to maintain a practical and reasonable way of encoding provisions with unlimited reentrancy into a finite model such as VASS. In the case of fully re-entrant provisions a special case of a finite state machine will be utilized. It will be called *counter automaton* and technically it can be viewed as regular finite state machine which additionally contains special symbols on states and transitions that are used to modify special counters. Conversely, ordinary finite state machines can be used for checking provisions which do not contain full reentrancy. The counter automata will be discussed first.

In the case of counter automata the interpretation of provisions is slightly modified in contrast with the classical closed model analysis. Simply put, a call sequence that may occur in parallel with other sequences with respect to a given provision will be bound to the thread that started it. In depth, a simultaneously regular expression can be viewed as a parallel composition of regular languages, i.e. sets of call sequences, specified by entirely regular subexpressions. Each thread is then supposed to issue method calls with respect to (at least) one of such regular expressions or remain silent with respect to the events being observed by this particular provision. In other words, method calls observed by the provision issued by a thread must form the empty word λ or a word matching a maximal entirely regular subexpression of the provision. If each thread in the model satisfies this condition and additionally the number of threads matching particular regular subexpression do not exceed upper parallel boundaries defined for the subexpressions (recall Definition 3.5), then the model is considered as valid with respect to the bad activity and the provision. Let's remark that the threads after the inline expansion are considered and the method calls and returns are represented by the labels on transitions. This approach enables mapping of the infinite nature of a provision with reentrancy to the infinite nature of the symbolic counter abstraction – number of threads.

Note that this interpretation of provisions is far more restrictive in comparison with the closed model analysis – while considering the *bad activity* property – and thus the incomplete model analysis will be on the safe side. The claim that any overall sequence of events valid under terms of incomplete model analysis will also be valid under terms of closed model analysis is affirmed by Theorem 3.8: if the sequence satisfies lower parallel guards, it is a direct consequence; otherwise if the sequence is valid in the first case, it is clearly a prefix of a trace of the provision and thus valid in the second case. On the other hand, consider a simple case where there are two threads which

together form a sequence of events matching a regular subexpression of a provision in the way that the first thread forms a prefix of the sequence and the second thread forms the rest. Let's assume that the threads are properly synchronized using locks. Such a construct will not violate the *bad activity* property in the closed model, whereas in the incomplete one it is easy to imagine a concrete situation in which it would.

As the above interpretation is more restrictive than the classical approach, it is natural to ask whether it might be somehow loosened. One possible approach would be to allow the threads to "reincarnate" – once a thread finishes a sequence that matches a regular subexpression, it might start another sequence, possibly matching a completely different subexpression. But this approach would no longer reside on the safe side since there would be "blind spots" in which it would be impossible to decide whether the thread had already started a new sequence or whether it was still continuing in the first one. Although results from such an interpretation might also be interesting, the safe and more restrictive approach will be focused on.

Counter Automata

As the property of TBP model imposed by the interpretation of provisions stated above is specified, it is necessary to introduce a form of observing mechanism which will evaluate the property during the checking of the model. An eventual violation of the property need to be projected into the reachability of a special purpose state, which will indicate the violation. The observing mechanism can be defined separately for each provision separately. This mechanism must fulfill two tasks induced by provision; (i) to track whether all threads issue call sequences that conform to individual maximal entirely regular subexpressions and (ii) to monitor the number of call sequences matching the individual subexpressions being issued in parallel.

The first task may be achieved using simulation of the finite state machine that accepts exactly union of the regular languages (given by the subexpressions) along the execution of any thread. The state of the automaton must be encoded into the thread-local state in order to follow the simulation within the Karp-Miller tree construction. If a thread-generated sequence of events does not conform to the automaton, then either the automaton will be required to perform a transition which is not defined or the thread will not reside in a state corresponding to a final state of the automaton as the thread terminates.

Let P_i be a simultaneously regular provision expression and let it contain $E_1, E_2, \dots, E_k$ maximal entirely regular subexpressions. Then let $\mathcal{A}_{P_i}$ be a finite state machine which accepts union of $E_1, E_2, \dots, E_k$ regular languages (it may be the automaton stated in Definition 3.9). Let bijection $c_{P_i} : Q_{\mathcal{A}_{P_i}} \to C_{P_i}$ map the states $Q_{\mathcal{A}_{P_i}}$ to a set of constants C_{P_i}. The set of constants C_{P_i} will then form a domain of the local variable that will represent the automaton state within the thread-local states. Thus let $q_{P_i,j}$ be a variable

that $Dom_{q_{P_i,j}} = C_{P_i}$ and $Init_{q_{P_i,j}} = c_{P_i}(s_0)$ where s_0 is the initial state of $\mathcal{A}_{P_i}$. Then let $q_{P_i,j}$ be added to the set of local variables of the j-th thread, The observing mechanism will then modify the value of $q_{P_i,j}$ according to $\mathcal{A}_{P_i}$ as the j-th thread issues an event which is in $filter_{P_i}$. Eventually, at the end of the execution of the thread, it will check whether the thread even issued an event observed by the provision and in that case whether $q_{P_i,j}$ correspond to a final state of $\mathcal{A}_{P_i}$. These variables will be added to all threads and the same mechanism will hold for every bound provision. In fact, the variables might be added only to those threads that may eventually issue an event observed by the provision. But for the sake of simplicity, let's add them to all the threads.

This observing mechanism may be implemented within EXPLORE procedure, however it is necessary to extend the model $\mathbb{P}$ with the set of bound provisions $P^b = \{P_1, P_2, ..., P_m\}$. Moreover, it is necessary to signal a violation of the provision so that it will be reflected in the reachability of thread states. For this purpose, let us define a fictive thread E with a single local state e and no outgoing transitions. The observing mechanism will instantiate thread E if there is no $\mathcal{A}_{P_i}$ transition according to current event or if $\mathcal{A}_{P_i}$ is not in a final or initial state once the appropriate thread ends. Note that if the automaton is in its initial state, it means that no event observed by the provision has been issued – of course only if the automaton does not contain any cycle which involves the initial state, but it can be assumed without loss of generality [2]. Reachability of the state e then reflects violation of the model property defined by the provisions with respect to (i). For clarity, the extended model $\mathbb{P}'$ then take the form $(\mathcal{G}, \bar{T}, \mathcal{R}, P^b)$ where $\bar{T} = \{(\bar{L}_1^{pc}, \Gamma_1^{pc}), (\bar{L}_2^{pc}, \Gamma_2^{pc}), ..., (\bar{L}_n^{pc}, \Gamma_n^{pc}), E\}$ and $\bar{L}_i^{pc} = L_i^{pc} \cup \{q_{P_1,i}, q_{P_2,i}, ..., q_{P_m,i}\}$. Algorithm 4 then presents the procedure that reflects provisions with respect to (i).

In order to monitor the number of thread instances that perform a sequence of events matching individual subexpressions of provisions (task (ii)), it is necessary to distinguish between particular subexpressions along the thread execution. A very similar problem has already been solved in generating components from unbound provisions. In that case, a special finite state machine was used, its states were related to groups of provision subexpressions and then tracked using counters. That approach can be reused here with minor modifications..

First, recall the automaton stated in Definition 3.9. This automaton clearly accepts a language formed by union of the maximal entirely regular subexpressions of the provision as the definition in fact describes disjunction of finite state machines and

[2]For every deterministic finite state machine there is an equivalent deterministic finite state machine that does not contain a cycle involving the initial state. It can be constructed by *pulling* the initial state out of the cycle, which is for example used in Definition 3.11.

Algorithm 4: The EXPLORE Procedure With Provisions

function EXPLORE $\left((\mathcal{G}, \bar{T}, \mathcal{R}, P^b),\ (s, L, f),\ l\right)$
- $Result \leftarrow \emptyset$
- **let** $\left(\bar{L}_i^{pc}, \Gamma_i^{pc}\right) \in T\ \ s.t.\ \ \psi(l) \in \Gamma_i^{pc}$ **begin** // pick the thread which l belongs to
  - **for** $(p_t, g_t, l_t, a_t, q_t) \in \delta_{\Gamma_i^{pc}}\ \ s.t.\ \ p_t = \psi(l)$ **do** // all transitions from $\psi(l)$
    - **if** VALUEOF $(g_t, s, l) = true$ **then** // is the guard is fulfilled?
      - $(s', l') \leftarrow$ ASSIGN $((s, l),\ a_t)$
      - $(s'', L_a, L_t) \leftarrow$ OBSERVE$^{(i)}$ $\left(P^b,\ (s', \{l\}, \{l'\}),\ l_t\right)$
      - add (s'', L_a, L_t) to $Result$
- **return** $Result$

function OBSERVE$^{(i)}$ $(Provisions,\ (s, L_a, L_t),\ Label)$
- $(s', L_a', L_t') \leftarrow (s, L_a, L_t)$
- **for** $P_i \in Provisions$ **do**
  - $L_t'' \leftarrow L_t'$
  - **for** $l \in L_t' \wedge l \neq e$ **do**
    - $l' \leftarrow l$
    - **if** $label \in filter_{P_i} \wedge label \neq \tau$ **then**
      - **let** $j\ s.t.\ pc_j \in dom(\gamma_l)$ **begin** // l belongs to j-th thread
        - $q \leftarrow c_{P_i}\left(\gamma_l\left(q_{P_i,j}\right)\right)$
        - **if** $\exists (q, Label, q') \in \delta_{\mathcal{A}_{P_i}}$ **then**
          - $(q, Label, q') \leftarrow$ a transition in $\delta_{\mathcal{A}_{P_i}}$ from q over $Label$
          - $(s, l') \leftarrow$ ASSIGN $\left((s, l),\ \left\{q_{P_i,j} \leftarrow c_{P_i}(q')\right\}\right)$ // s unchanged
          - replace l with l' in L_t''
        - **else**
          - add e to L_t''
    - **if** THREADEND $(l') = true$ **then** // the thread terminates in l'
      - $q \leftarrow c_{P_i}\left(\gamma_l\left(q_{P_i,j}\right)\right)$
      - **if** $q \notin F_{\mathcal{A}_{P_i}} \wedge q \neq$ *initial state of* $\mathcal{A}_{P_i}$ **then**
        - add e to L_t''
  - $L_t' \leftarrow L_t''$
- **return** (s, L_a, L_t')

subsequent determinization. Additionally, for the purpose of task (ii) it is necessary to count also the first automaton's transition taken by a thread, as the total number of threads that participate on any of the subexpressions may be limited by the upper parallel boundaries. As the definition allows the initial state to be part of a cycle, which makes impossible to distinguish whether an automaton's transition from the initial state correspond to the first event issued by the thread, it is necessary to *pull* the initial state out of the cycle. Definition 3.11 presents a version of the automaton with *pulled* initial state.

Definition 3.11. Let P be a simultaneously regular provision expression and $E_1, E_2, \dots, E_k$ be the maximal entirely regular subexpressions of P. Let $\mathcal{A}_i$ denote a deterministic finite state machine representing E_i. Without loss of generality let $\forall i, j \in \{1,2,\dots,k\} : Q_{\mathcal{A}_i} \cap Q_{\mathcal{A}_j} = \emptyset$

Then $O_P = (Q, \Sigma, \delta, s_0, F)$ is a finite state machine such that

- the set of states $Q \subseteq \wp\left(\bigcup_{i=1}^{k} Q_{\mathcal{A}_i}\right)$;
- the alphabet $\Sigma = \bigcup_{i=1}^{k} \Sigma_i$, where Σ_i denotes alphabet (input symbols) of $\mathcal{A}_i$;
- the transition relation $\delta = \delta' \cup \delta^0$ where δ' is defined as

$$\begin{aligned}(S,x,T) \in \delta' \quad \iff \quad & S \neq \emptyset \wedge T \neq \emptyset \\ \wedge \quad & \forall s \in S \, \forall i \in \{1,2,\dots,k\} : (s,x,t) \in \delta_{\mathcal{A}_i} \Rightarrow t \in T \\ \wedge \quad & \forall t \in T \, \exists s \in S \, \exists i \in \{1,2,\dots,k\} : (s,x,t) \in \delta_{\mathcal{A}_i}\end{aligned}$$

  and δ^0 is defined as

$$(\emptyset, x, T) \in \delta^0 \iff T \neq \emptyset \wedge (S,x,T) \in \delta'$$

  where S is the set of initial states of $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_k$;
- the initial state s_0 is $\emptyset$;
- the set of final states $F \subseteq Q$ is defined as follows:

$$S \in F \iff \exists s \in S \, \exists i \in \{1,2,\dots,k\} : s \in F_{\mathcal{A}_i}$$

Recall the relation between the states of the automaton and entirely regular subexpressions of the provision expression. A state is said to *overlap* a subexpression E_i if it contains a state of the subexpression's automaton $\mathcal{A}_i$. The relation was characterized with function $\mathcal{C}_P : Q_{\mathcal{D}_P} \to \wp(\{1,2,\dots,k\})$ and the $\mathcal{F}_P : Q_{\mathcal{D}_P} \to \wp(\{1,2,\dots,k\})$ expressed it with respect to the final states (see equations 3.10 and 3.11). This concept will be used for O_P as well, but formally it must be adjusted for the *pulled* initial state

in order to make it equivalent to the original initial state. Therefore let S be the original initial state, i.e. a set of initial states of automata $\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_k$, then $\mathcal{C}_P(\emptyset) := \mathcal{C}_P(S)$ and $\mathcal{F}_P(\emptyset) := \mathcal{F}_P(S)$. Recall also that a transition is called *specializing* if it starts at a state which overlaps with more subexpressions than the target state, i.e. it enters a more specialized part of the automaton. Note that Lemma 3.10 holds for this automaton as well.

As with the case of environment generating, the observing mechanism will also count number of threads taking a specializing transition. However, in contrast to the environment generating, the counters will be represented by fictive threads with single states – a fictive thread for each counter. Incrementing the counter is then equivalent to launching a new instance of such a fake thread. Determining whether a counter can eventually exceed a certain limit m is then equivalent to finding out whether a configuration where the appropriate thread state is occupied at least $(m+1)$-times is reachable.

More concretely, let P_i be a simultaneously regular provision expression and O_{P_i} be a deterministic finite sate machine according to Definition 3.9. Let $\mathcal{C}_{P_i}\left(Q_{O_{P_i}}\right)$ denote the range of the function $\mathcal{C}_P$ and $\mathcal{F}_{P_i}\left(Q_{O_{P_i}}\right)$ denote the range of $\mathcal{F}_P$. For each R an element of $\mathcal{C}_{P_i}\left(Q_{O_{P_i}}\right) \cup \mathcal{F}_{P_i}\left(Q_{O_{P_i}}\right)$ let $C_{P,R}$ denote the fictive thread representing the counter for R and let $l_{C_{P,R}}$ denote the only local state of $C_{P,R}$.

When an automaton associated with a thread takes a specializing transition from p to q then R counter will be incremented if $R \subset \mathcal{C}(p)$ and $\mathcal{C}(q) \subseteq R$, which in other words says that all counters that might be on a way from p to q with respect to specialization will be incremented. Additionally, when a thread terminates and the observing automaton is in a final state q, then R counter will be incremented if $R \subset \mathcal{C}(q)$ and $\mathcal{F}(q) \subseteq R$. This is equivalent to a specializing transition to a state overlapping with just the subexpressions that are accepted by the final state q – since it is known that the thread terminates and it will no longer issue any event, such a fictive transition may be taken. Algorithm 5 presents the OBSERVE procedure which increments the counters. Using this procedure within EXPLORE in Algorithm 4 fulfills both the tasks (i) and (ii).

From the above, it follows that the R's counter value indicates the number of threads that have already issued a sequence of events which may be eventually extended to a word matching a subexpression E_i such that $i \in R$ and such a word exists for all $i \in R$. In other words, any of those threads issued a sequence u such that for every $i \in R$ there is a word v matching E_i and u is a prefix of v. Thus if a configuration in which the counter exceeds the sum of upper parallel bounds of the subexpressions indexed with R is reached, it is necessary that the provision will eventually be violated. Simply put, any possible extension of the words will form a parallel interleaving of too many

Algorithm 5: The OBSERVE Procedure With Counters

function OBSERVE$^{(i+ii)}$ $(Provisions,\ (s, L_a, L_t),\ Label)$
- $(s', L'_a, L'_t) \leftarrow (s, L_a, L_t)$
- **for** $P_i \in Provisions$ **do**
  - $L''_t \leftarrow L'_t$
  - **for** $l \in L'_t \land l \neq e$ **do**
    - $l' \leftarrow l$
    - **if** $label \in filter_{P_i} \land label \neq \tau$ **then**
      - **let** j *s.t.* $pc_j \in dom(\gamma_l)$ **begin** // l belongs to j-th thread
        - $q \leftarrow c_{P_i}\left(\gamma_l\left(q_{P_i,j}\right)\right)$
        - **if** $\exists (q, Label, q') \in \delta_{\mathcal{A}_{P_i}}$ **then**
          - $(q, Label, q') \leftarrow$ a transition in $\delta_{\mathcal{A}_{P_i}}$ from q over $Label$
          - $(s, l') \leftarrow$ ASSIGN $\left((s, l),\ \left\{q_{P_i,j} \leftarrow c_{P_i}(q')\right\}\right)$ // s unchanged
          - replace l with l' in L''_t
          - $L''_t \leftarrow L''_t \cup$ COUNTERS $(P_i,\ \mathcal{C}_{P_i}(q),\ \mathcal{C}_{P_i}(q'))$
        - **else**
          - add e to L''_t
    - **if** THREADEND $(l') = true$ **then** // the thread terminates in l'
      - $q \leftarrow c_{P_i}\left(\gamma_l\left(q_{P_i,j}\right)\right)$
      - **if** $q \notin F_{\mathcal{A}_{P_i}} \land q \neq$ *initial state of* $\mathcal{A}_{P_i}$ **then**
        - add e to L''_t
      - **else**
        - $L''_t \leftarrow L''_t \cup$ COUNTERSINC $(P_i,\ \mathcal{C}_{P_i}(q),\ \mathcal{F}_{P_i}(q))$
  - $L'_t \leftarrow L''_t$
- **return** (s, L_a, L'_t)

function COUNTERSINC $(P, Over_{src}, Over_{tgt})$
- $Result \leftarrow \emptyset$
- **for** $R \in \mathcal{C}_P(Q_{O_P}) \cup \mathcal{F}_P(Q_{O_P})$ **do** // iterate over all possible counters
  - **if** $Over_{tgt} \subseteq R \subset Over_{src}$ **then**
    - add $l_{C_{P,R}}$ to $Result$
- **return** $Result$

words of a subexpression – if only the valid extensions are considered (as the invalid cause violation immediately). Let the maximal valid value of a counter $C_{P,R}$ be denoted $\mathcal{M}_{C_{P,R}}$.

$$\mathcal{M}_{C_{P,R}} = \sum_{i \in R} \mathcal{U}_P(E_i)$$

In summary, if a configuration (s, L, f) such that $e \in L$ or any counter is exceeded (3.13) then the model violates the *bad activity* property.

$$\exists P_i \in P^b \; \exists R \in \mathcal{C}_{P_i}\left(Q_{O_{P_i}}\right) \cup \mathcal{F}_{P_i}\left(Q_{O_{P_i}}\right) \text{ s.t. } l_{C_{P_i},R} \in L \wedge f\left(l_{C_{P_i},R}\right) > \mathcal{M}_{C_{P,R}} \tag{3.13}$$

The careful reader will have surely noticed that the lower parallel guards have not been considered, in contrast with the environment generating. Imagine the checking of the lower parallel guards in the case of unlimited number of threads. At any point, even if a lower parallel guard does not hold, there are always infinitely many threads and each of them may eventually trigger some events that amend the violated lower parallel guard. Therefore, the only point where it is reasonable to take the lower parallel guards into account is the point of termination of the last thread. But such point does not exist in an environment with unlimited number of threads as new thread instance can always be started. In fact, the lower parallel guard reflects rather *no activity* property in this case. Since this work focuses on the models with unlimited number of threads with respect to *bad activity*, it is convenient to omit the lower guards in the analysis.

On the other hand, consider a case where a bound provision does not contain full reentrancy operator. The model still triggers unlimited number of thread instances as an unbound provision contains full reentrancy, but the bound provision is suddenly expressible with finite means – deterministic finite state machines. Such state machines would be simulated along execution of the model in a very similar way as the provision automata specified above. The main difference is such that this state machine would be simulated from a global perspective, for all threads at once. Then the states of the machine would be encoded into global variables instead of local ones. The OBSERVE procedure for such machines is presented in Algorithm 6. Note that the point where a thread terminates is no longer considered. The reason is, that another thread may always change the state of the machine as the machine is simulated globally.

The main benefit of using deterministic finite state machines over the provision automata as specified above is that it does not introduce spurious errors while it is clearly safe. Consider an incomplete model which does not contain full reentrancy on any bound interfaces and assume that the generated environment does not produce errors (the LTSA state e contained in the environment is not reachable in any thread). Then as the finite state machines accept exactly the allowed traces of the provisions,

for every bad activity error detected within the incomplete model there is a valid environment containing a finite number of threads which generates the error trace in the closed model. On the other hand, let's take an environment with k threads that is valid and composition with the incomplete model produces bad activity error detected in the closed model checking. Such an environment, as it is valid, produces a subset of the traces produced by the generated environment. Thus the bad activity trace is also present in the incomplete computation. Consequently there is reachable configuration from which a thread emits an event (over method m) for which the current state of a finite state machine that accepts P_i s.t. $m \in filter^{P_i}$ does not have any outgoing transition.

Algorithm 6: The OBSERVE Procedure For Provisions Without Full Reentrancy

function OBSERVE$^{(FSM)}$ $(Provisions,\ (s,L_a,L_t),\ Label)$
- $(s',L'_a,L'_t) \leftarrow (s,L_a,L_t)$
- **for** $P_i \in Provisions$ **do**
  - **if** $label \in filter_{P_i} \wedge label \neq \tau$ **then**
    - $q \leftarrow c_{P_i}(\gamma'_s(q_{P_i}))$ // q_{P_i} is a global variable that encodes the state of $\mathcal{FSM}_{P_i}$
    - **if** $\exists (q, Label, q') \in \delta_{\mathcal{FSM}_{P_i}}$ **then**
      - $(q, Label, q') \leftarrow$ a transition in $\delta_{\mathcal{FSM}_{P_i}}$ from q over $Label$
      - $(s'',l) \leftarrow$ ASSIGN $((s',\emptyset),\ \{q_{P_i} \leftarrow c_{P_i}(q')\})$
      - $s' \leftarrow s''$
    - **else**
      - add e to L'_t
- **return** (s',L_a,L'_t)

Additionally, there might be a possibility of reducing spurious errors induced by counter automata in their cooperation with ordinal finite state machines being simulated globally as for provisions without full reentrancy. Some key ideas of such approach will be outlined in following section. For the sake of clarity, the globally observed finite state machines will from here on be called *observers*.

Combining Counter Automata and Observers

The idea of combining the two approaches in order to reduce spurious errors caused by counter automata is based on several simple observations. Note that in the following text just key observations and ideas are merely outlined and indeed some of them would deserve more detailed argumentation or proofs, but these are beyond the scope of this work. The ideas are presented in order to illustrate available possibilities to reduce the spurious errors, which might be eventually a subject of further work.

First, the provisions that contain full reentrancy may also contain some subexpressions that are not allowed to be executed in fully re-entrant manner. Such provisions checked by counter automata produce spurious errors that might have been avoided if the not fully re-entrant subexpression were checked separately using an observer. Consider the simple example $(m()*)||((n()*)|*)$. If there were a way to split the provision into two parts, one that might be checked using observer and the other, containing the full reentrancy, checked by the provision automaton, then it might produce less spurious errors. Note that concretely the $m()*$ part causes spurious error even for such basic cases, as it is using `sync` to synchronize calls m from a reaction to a fully re-entrant provided method.

Note also that some spurious errors might be avoided using $(Q;Q*)|*$ instead of $Q|*$. These two provisions are equivalent in terms of allowed traces; $traces^{(Q;Q*)|*} = traces^{Q|*}$, but the former is far less restrictive under the counter automata interpretation. This fact might be utilized to partially reduce spurious errors even in the case of pure counter automata. Unfortunately, we are not aware of any similar construct applicable for limited reentrancy or parallel in general.

It is worth mentioning the fact that operators $|$ and $||$ are interchangeable in simultaneously regular provisions if considering just bad activity error. Recall that the bad activity is defined using prefixes of traces. Let's define $prefix^P$ as a set of sequences of events where each sequence is a prefix of a sequence in $traces^P$. Then clearly for any simultaneously regular provision P holds $prefix^P = prefix^{P^|} = prefix^{P^{||}}$, where $P^|$ denotes a provision expressions derived from P where all occurrences of $||$ were replaced with $|$ and vice-versa for $P^{||}$. Thus any simultaneously regular provision can be flattened into form $Q_1||Q_2||...||Q_k$ where no Q_i contains $|$ or $||$ operators.

Another useful fact is that provisions operating on the same set of methods together form the intersection of their allowed traces. Consider set of provisions $P_1, P_2, ..., P_n$ in a model M. Let $traces^M$ denote a set of all allowed traces in the model M and $traces^M_\Sigma$ is a set of all non-empty sequences s of symbols not in $\Sigma^{\uparrow/\downarrow}_{Dom_E}$ for which exists a an arbitrary sequence s' s.t. $s \otimes s' \cap traces^M \neq \emptyset$. Simply $traces^M_\Sigma$ correspond $traces^M$ where all symbols that are not in $\Sigma^{\uparrow/\downarrow}_{Dom_E}$ were erased. If $filter^{P_i} = filter^{P_j}; i \neq j$, then

$$traces^M_{filter^P_i \cup filter^P_j} \subseteq traces^{P_i} \cap traces^{P_j}$$

Additionally, if for all P_k s.t. $k \neq i \land k \neq j$ holds that $filter^{P_k} \cap filter^{P_i} = \emptyset$, then

$$traces^M_{filter^P_i \cup filter^P_j} = traces^{P_i} \cap traces^{P_j}$$

Note that this fact was also utilized for creating finite state machine for provision driven computation in open model analysis in [Poc10].

From the above follows, that if P and Q are two provisions operating on mutually disjunct sets of methods, then the provision $P||Q$ can be split into two provisions $P||\left(Any_{filter^Q}\right)$ and $\left(Any_{filter^P}\right)||Q$ where Any_Σ represents a fictive provision that allows all possible words over symbols of $\Sigma_{Dom_E}^{\uparrow/\downarrow}$. The intersection of these two will form $traces^{P||Q}$. Let P do not contain full reentrancy, then the provision $P||\left(Any_{filter^Q}\right)$ can be checked using observer, as Any_Σ can be expressed as a repetition (*) of alternative (+) of all the symbols in $\Sigma_{Dom_E}^{\uparrow/\downarrow}$, which is a regular expression, thus it is possible to create such a finite state machine. On the other hand, Any_Σ can also be expressed as fully re-entrant repetition (*) of alternative (+) of all the symbols in Σ_{Dom_E}, which is a simultaneously regular expression and it is possible to create an counter automaton for it. Thus if P contains full reentrancy, then $P||\left(Any_{filter^Q}\right)$ can be checked using counter machine.

So far, this can give an opportunity to split some fully re-entrant provisions into several parts and the parts containing full reentrancy can be checked using counter automata, whereas parts without full reentrancy can be checked by observers. However, if all the parts contain full reentrancy, there is no benefit in splitting the provision, as all the parts must be still checked using counter automata. Note also, that the condition on the parts, that they must operate on mutually disjunct sets of methods can be refined so that they must operate on disjunctive sets of parametrized labels. Then, for example, the provision expression $((\mathtt{m(TRUE)}|*) : \mathtt{TRUE})||(\mathtt{m(FALSE)} : \mathtt{FALSE})$ can be split as well.

4. Prototype Implementation

Part of this thesis is a prototype implementation of the incomplete TBP model analysis. The tool is a part of a larger whole – TBP toolchain, which currently consists of a library for parsing and converting TBP specifications to a representation of TBP models (named *TBPLib*) and a module for performing analysis over TBP models (named *Badger*). Except for the introduced prototype, the toolchain already contains a tool for checking closed TBP models.

In this chapter, the structure of the TBP toolchain will be briefly described with emphasis on the parts that were developed for the prototype. The structure overview can serve as a very brief guide to the sources presented on the attached CD ROM (see Appendix A). However, the CD ROM contains also generated class documentation, which better suits this purpose. Then the process of checking the incomplete TBP model will be outlined as it is performed by the prototype.

4.1 Structure of the TBP Toolchain

As was already mentioned, the toolchain consists of TBPLib and Badger, which are two separate projects. It is also worth mentioning, that they are both implemented in Java.

4.1.1 TBPLib

The overall functionality of the library includes parsing of TBP specifications and converting them to a representation of TBP models, including LTS parts and deterministic finite state machines for observing provisions. During implementation of the prototype for this thesis, the library was extended with a representation of counter automata, a module for generating artificial environment in form of LTSA and the inline expansion over LTSA. A special LTSA representation was added, which suits well for the inline expansion and also covers the extension, that more than one assignment can be performed on a transition. Table 4.1 presents an overview of crucial packages of the library with brief description. Packages that were introduced or non-trivially modified during the implementation of the prototype are highlighted.

4.1.2 Badger

The project, where the TBP model checkers are stored is named Badger. Currently it contains a tool for closed TBP model analysis and the prototype of incomplete TBP

package name relative to the root package `org.ow2.dsrg.fm.tplib`	**description**
`.architecture`	Provides means for capturing component architecture related to the TBP specifications.
`.event`	Encoding of parametrized events on provisions.
`.ltsa`	Contains classes and interfaces representing LTS and finite state machines. The extended LTSA representation was added.
`.ltsa.visitor`	Contains visitors operating on elements of the previous package. The in-line expansion is here implemented in class `InlineMethodCallsVisitor`.
`.node`	Contains representation of nodes of a syntactical tree of the TBP specification.
`.parser`	Contains parser of TBP specifications generated using ANTLR.
`.provision`	Contains representation of counter automata.
`.provision.actuator`	Contains classes responsible for the environment generation.
`.provision.actuator.lpg`	Provides tools for working with lower parallel guards.
`.reference`	Contains classes for capturing references in the specifications (method, type, variable and constant names).
`.util`	Contains utility classes (writing LTSA and FSM as DOT graphs) and a main helper which provides simplified utilization of the most common operations provided by the library (`TransformationChain`).

Table 4.1: Main structure of TBPLib

package name relative to the root package `org.ow2.dsrg.fm.badger`	**description**
`.ca`	Contains the tool for analysing incomplete TBP models using symbolic **c**ounter **a**bstraction.
`.ca.karpmiller`	Contains an representation of the Karp-Miller tree and an implementation of its construction procedure.
`.ca.statespace`	Provides representation of local and global states and the state space exploration phase (the EXPLORE procedure).
`.simulation`	Closed model analysis – provides interfaces to be implemented by a *behavioral interpreter*, which is a Java code generated from TBP model.
`.translation`	Closed model analysis – provides tools for generating the behavioral interpreters.
`.traversal`	Provides means for traversing the behavioral interpreters.

Table 4.2: Main structure of Badger

model analysis using symbolic counter abstraction.

The prototype is mainly implemented in `org.ow2.dsrg.fm.badger.ca` package, where is also the entry point to the application – the *main* class `Check`. Simply put, it transforms input TBP specifications into a representation of the incomplete TBP model, converts the bound provisions to counter automata or observers and generates environment from the unbound provisions using TBPLib. Then it starts the Karp-Miller tree construction and checks whether erroneous configurations are reachable. If there is an erroneous configuration reachable, it prints the path from root to the erroneous configuration to a file as a DOT graph.

Table 4.2 presents overview of crucial packages of the library with brief description.

4.2 Limitations

The proof-of-concept prototype implements the key issues of the proposed analysis. It provides a functional solution for the analysis of TBP specifications that represent incomplete TBP models. However, in the following text are discussed the main limitations of the implementation.

First, the implementation is not mainly focused on optimality in terms of performance. Although some minor optimizations of the Karp-Miller tree construction pro-

posed in [Zha09] were implemented, there still remains a vast space for further optimizations. Mainly the representation of the VASS configurations and the implementation of the state space exploration phase might be a subject of further optimizations.

Within the process of generating artificial environments, the lower parallel guards are currently not fully implemented. As presented in the section 3.2.1, the lower parallel guards, which are propositional formulae, must be derived from the provisions, converted into the disjunctive normal form and then expanded using available counters. Currently, the conversion into the disjunctive normal form is not implemented[1]. Thus the lower parallel guards are constructed as if the provisions contained the and-parallel operators in place of the actual or-parallel operator occurrences. Such lower parallel guards are then naturally in the disjunctive normal form and thus they may be directly expanded. Note that this may affect the environments in the way that they generate only a subset of the event sequences induced by the provisions.

Also the output of the program in case of a detected error, the error trace, should be enhanced in order to make the tool practically usable for identifying real bugs in the specifications. Currently, the tool writes the path of the Karp-Miller tree which leads to the error into a file in the form of DOT graph. Although this path coupled with DOT graphs of the LTSAs (which the tool optionally generates) allows a full reconstruction of the sequence of the statements which lead to the error, such a reconstruction is tedious and it could be performed automatically by the tool. For this purpose, a mapping of LTSA transitions on appropriate statements in TBP specification should be implemented. Currently, there is a skeleton of such a mapping, but it does not cover the renaming of variables during the in-line expansion.

On the other hand, it is worth mentioning, that the tool supports two different interpretations of provision expression. The reason is that the TBPLib library originally interpreted $A|k$ as $A*|A*|...|A*$ and $A|*$ as $A*|A*....$ But in this work, the interpretation $A|k = A||A||...||A, A|* = A||A||...$ is strictly followed since the former can be simply expressed within the later using $(A*)|k$ and $(A*)|*$. However, in order not to break other tools that use TBPLib, the behavior could not be simply changed. Instead, the classes responsible for generating finite state machines from provision expressions were parametrized. Additionally, in order to make possible to keep consistency between the closed model checker and the incomplete model checker, the reentrancy interpretation can be changed using `-rr` command-line option.

[1]It is worth mentioning that at least one algorithm solving this issue is well known.

Conclusion

In this thesis, a special scenario of analysis of TBP specifications involving unlimited number of threads was investigated. A particular class of TBP models, called incomplete TBP models, was identified. Then a safe analysis of correctness of the incomplete TBP models under any environment that meets the models provisions was designed. The correctness is perceived with respect to the presence of bad activity errors.

The analysis deals with infinite state space using symbolic counter abstraction. This abstraction reduces the state space symmetries caused by unbounded thread replication by using counters of the threads residing in individual thread-local states (valuations of local variables). The thread state reachability problem is then reduced to the VASS coverability problem, which is solved by using the Karp-Miller tree construction.

The symbolic counter abstraction interleaves the Karp-Miller tree construction with context-aware state space exploration phase. It prevents the necessity of explicit construction of the VASS and tackles the problem known as local state space explosion. In this work, the state space exploration phase for the TBP models is designed.

Moreover, the bound provisions (provisions over methods used internally in the model) need to be represented as the state reachability properties acceptable for the VASS coverability problem. This is the main challenge of the analysis and it is very difficult if even possible in general. This work presents a solution for a specific class of provisions with full reentrancy and additionally states a solution for provisions without full reentrancy.

For the provisions with full reentrancy a special case of finite state machines (called counter automata in this work), which utilize the thread counters to capture parallel nature of the provisions, was designed. However, the analysis using these automata unfortunately suffers from the possibility of spurious errors. On the other hand, for the provisions without full reentrancy, ordinary finite state machines can be utilized and these do not induce spurious errors at all. Additionally, an approach to further reduce the spurious errors induced by the former case by using a combination of the two state machines was outlined.

In order to perform the incomplete model analysis, which takes into account any environment that fulfils the unbound provisions of the model (provisions over provided methods), it is necessary to create an artificial environment that is able to issue any combination of events allowed in the provisions. Such an environment consists of threads derived from the provisions in a way, which assures all possible executions of the threads to eventually issue all the allowed sequences of events. As the provisions are supposed to contain full reentrancy, the number of the instances of these

threads is unlimited. The threads cooperate by using special counters, which assure that interleavings of event sequences generated by the individual threads stay within the boundaries given by the provision.

Unfortunately, the artificial environment created this way can possibly introduce spurious errors in some cases of provisions that distinguish return values of method calls and do not accept all the possible values at the same time. However, if such (possibly spurious) error occurs, it is identified early in the environment. Additionally, as such a spurious error is related to the return value of a provided method, it often signals that the provisions do not properly capture the internal behavior of the model and should be reformulated.

As a part of this thesis, a proof-of-concept prototype was developed. The main purpose of the prototype is to capture all the key aspects of the proposed analysis. As the performance is not the main goal of the implementation, there is a lot of opportunities for further optimizations. The optimization of the algorithms and implementation is an open issue which needs to be addressed in order to make the analysis applicable to large real-life TBP specifications.

Beyond the scope of the incomplete TBP model analysis, the key ideas of the proposed analysis might potentially be used for closed model analysis with unlimited number of threads. Thus this work also presents a small step towards allowing unlimited thread replication or dynamic thread instantiation in the TBP specifications. Also, some aspects of the proposed analysis might possibly be useful for extending the open model analysis to an unlimited number of threads.

Bibliography

[Bae05] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, 2005.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36:1257–1284, September 2006.

[BHK+10] Gérard Basler, Matthew Hague, Daniel Kroening, C.-H. Luke Ong, Thomas Wahl, and Haoxian Zhao. Boom: Taking boolean program model checking one step further. In *TACAS'10*, pages 145–149, 2010.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. *Software Engineering Research, Management and Applications, ACIS International Conference on*, 0:40–48, 2006.

[BMWK09] Gerard Basler, Michele Mazzucchi, Thomas Wahl, and Daniel Kroening. Symbolic counter abstraction for concurrent software. In *Proceedings of CAV 2009*, volume 5643 of *LNCS*, pages 64–78. Springer, 2009.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26:109–120, September 2001.

[ET99] Allen E. Emerson and Richard J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *CHARME'99*, pages 142–156, 1999.

[GG97] Richard Grimes and Dr Richard Grimes. *Professional Dcom Programming*. Wrox Press Ltd., Birmingham, UK, UK, 1997.

[Hol03] Gerard Holzmann. *The Spin model checker: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[HP76] John E. Hopcroft and J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. Technical report, Ithaca, NY, USA, 1976.

[KM69] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, May 1969.

[KPS09] Jan Kofron, Tomas Poch, and Ondrej Sery. Tbp: Code-oriented component behavior specification. In *Software Engineering Workshop; Proceedings: IEEE Software Engineering Workshop (32nd: C008: Kassandra, Greece)*, pages 75–83. IEEE Computer Society Press, 2009.

[Poc10] Tomas Poch. *Towards Thread Aware Component Behavior Specification*. PhD thesis, Charles University in Prague, Department of Distributed and Dependable Systems, 2010.

[Sak09] Kenneth Saks. *JSR 318: Enterprise JavaBeans(tm) 3.1*, 2009.

[Zha09] Haoxian Zhao. Using the karp-miller tree. construction to analyse. concurrent finite-state programs. Master's thesis, University of Oxford, Kellogg College, Oxford, UK, 2009.

Appendix A
Content of the enclosed CD ROM

This thesis is accompanied by the CD ROM containing source code and binaries of the prototype implementation. The CD ROM is organized as follows:

`/readme.txt`	Brief description of the CD ROM content.
`/dist/`	Binary distribution of the TBP toolchain including the prototype. Contains java archives of TBPLib and Badger. This directory also contains generated class documentation in HTML format.
`/doc/`	Contains electronic version of this thesis.
`/src/`	Source code of the TBP toolchain including the prototype and some example TBP specifications.