On Structural Aspect of Parallel Maximal Activities in Enriched Robustness Diagram with Loop and Time Controls

Edu S. Petilos[1*], Richelle Ann B. Juayong[1], Jasmine A. Malinao[2], and Francis George C. Cabarle[1]

[1] Department of Computer Science, College of Engineering, University of the Philippines Diliman, Quezon City, Philippines,
{espetilos, rbjuayong, fccabarle}@up.edu.ph
[2] Division of Natural Sciences and Mathematics, University of the Philippines Tacloban College, Tacloban City, Philippines,
jamalinao1@up.edu.ph
*Corresponding author

Abstract. In this study, we introduce a parallel implementation for generating sets of maximal activities for workflows or systems modeled using the Enriched Robustness Diagram with Loop and Time Controls (ERDLT). This is inspired by literature on its predecessor multidimensional workflow known as Robustness Diagram with Loop and Time Controls (RDLT), but with considerations on control structures that are unique to the former. Specifically, we propose a modification to the generation of traversal tree algorithm for RDLTs to consider the additional components of ERDLTs, including some suggestions for its implementation in parallel code. We included theorems showing the consistency of the modified algorithm with its original version in terms of producing results similar to the activity extraction algorithm for ERDLTs. We found that unlike an RDLT having only one traversal tree, the presence of guards implies multiple traversal trees for the same ERDLT model, which have implications on its parallel activities.

Keywords: Modeling, Parallel Activities, RDLT, Structural Profiles, Workflow Models

1 Introduction

The Robustness Diagram with Loop and Time Controls (RDLT) [1] is a graphical model used to represent workflows. To define a sequence of steps from a source to a sink node in a modeled workflow, a graph traversal algorithm similar to a depth-first graph search, called the activity extraction algorithm (or Algorithm $\mathcal{A}$), is employed. The output of this algorithm, and hence the extracted sequence of steps, is called an **activity** from the modeled system. There can be multiple distinct activities derivable from the same RDLT model, and Algorithm $\mathcal{A}$ can extract only one activity at a time.

J. Caro et al. (eds.), *Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2025)*, Atlantis Highlights in Computer Sciences 24,
https://doi.org/10.2991/978-94-6239-638-8_15

In [2], new concepts are established that were used to define multiple activities that can be executed in parallel in some RDLT model, called **parallel activities**. It included algorithms for generating traversal trees and for extracting parallel activities from RDLT models, where the former focuses on the structural aspect while the latter includes the behavioral aspect. In other studies, [3,4] sought to extend RDLT to address its limitations in terms of the representation of some object-oriented concepts incorporated into some workflow concepts. For this reason, new vertex and arc types are proposed for RDLTs that led to the conceptualization of its new variant, called the Enriched RDLT (ERDLT). Workflow or system modeling with ERDLT was shown to be improved in terms of having representations of more specific structures and relationships, such as object instance handling, inheritance, part-whole relationships and ownerships, mutually exclusive paths, and multi-module interactions among others.

With recent developments related to parallel activities for RDLT, it is a natural question to also reflect on how parallel activities can be identified and extracted in ERDLT which will ultimately extend ERDLT modeling with parallel workflows. To explore this, this paper aims to establish the relevant concepts to formalize parallel activities in ERDLT, including an algorithm to generate traversal trees from ERDLT models and design suggestions for its parallel code implementation. We deal with theoretical results only, and hence we suggest producing experimental results and validation as part of future work.

The rest of the paper is structured as follows. Section 2 entails the preliminaries and Section 3 summarizes the problem and goals of this study. Section 4 presents our methodology, while the results are discussed in Section 5. Finally, Section 6 contains the conclusions and recommended future work of this study.

2 Preliminaries

This section contains the relevant concepts used in the study presented in this paper. Section 2.1 discusses the components of RDLT, followed by the description of activity profiles and the workflow patterns of RDLT models in Sections 2.2 and 2.3, respectively. The parallel activities in RDLT and its underlying concepts are discussed in Section 2.4, whereas ERDLT is introduced in Section 2.5.

2.1 Robustness Diagram with Loop and Time Controls

The RDLT is a novel concept introduced in [1]. It is a multidimensional workflow model capable of modeling complex systems in their graphical representation. Some real-world systems that were able to be represented using this model, including an adsorption chiller system presented and analyzed in [1]. Use case texts were also modeled using RDLT such as a use case for a library system illustrated in [3], shown in Figure 1.

The vertex set V of an RDLT model R is composed of three types. **Entity objects** represent internal components of the modeled system, such as the classes found in its domain model. On the other hand, **boundary objects** depict

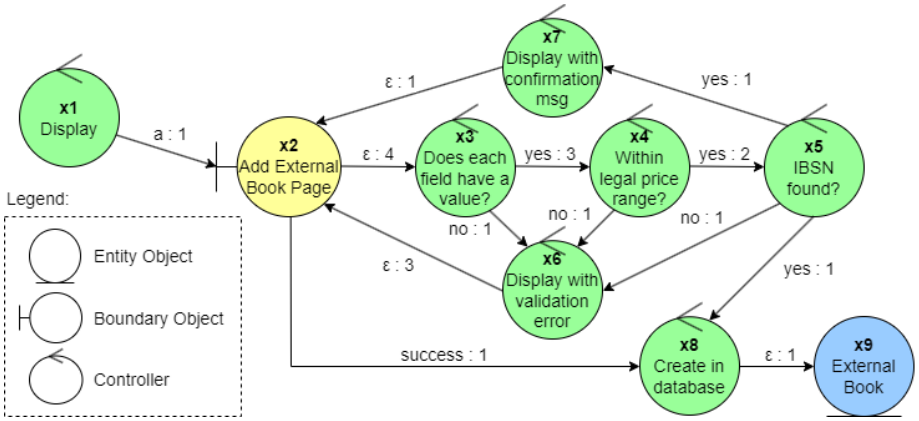

Fig. 1. Add External Books to Catalog Use Case RDLT Model (from [3])

an interface for users to interact with the system, while **controllers** illustrate system functions and logic implemented in the system [1,5]. A directed arc is used to connect these vertices in an RDLT model, where each such arc in the set E of R has two static attributes, namely the **constraints** (C-attribute; see left-hand arc values) and **limits** (L-attribute; see right-hand arc values). When a node in an RDLT model does not have incoming(outgoing) arcs, it is called a **source**(**sink**) vertex [1]. Generally, it is where the system process starts(ends). For brevity, we exclude the substructure of RDLT with a unique behavior in this section. In [3], an RDLT model without this substructure is referred to as a *simple RDLT*. On the left of Figure 2 illustrates an arbitrary simple RDLT model R. The static attributes of the arcs are written in the form $C : L$, where $C = \epsilon$ means that the arc does not have constraints, while the values of L can only be positive integers. An activity can be derived from R by performing a traversal from the source $x1$ to the sink $y4$, where each time step of traversal per arc is recorded sequentially as shown on the right of Figure 2. We discuss RDLT activities in the next subsection.

2.2 Activities in RDLT Models

An **activity** of an RDLT is a sequence of steps that can be derived from its source to a sink. An RDLT model R of a system can have multiple distinct activities depending on the nodes that have been visited. In [1], the **activity extraction algorithm** (or *Algorithm* $\mathcal{A}$) is used to traverse R to extract possible activities that can be derived from the system being modeled. It uses a depth-first search approach in which the availability of the arcs for traversal depends on their C- and L-attribute values that represent the constraints and limits, respectively. In RDLT, a constraint associated with an arc (x, y) is a parameter supplied by node x to node y. In order to visit a node, all constraints from its incoming arcs

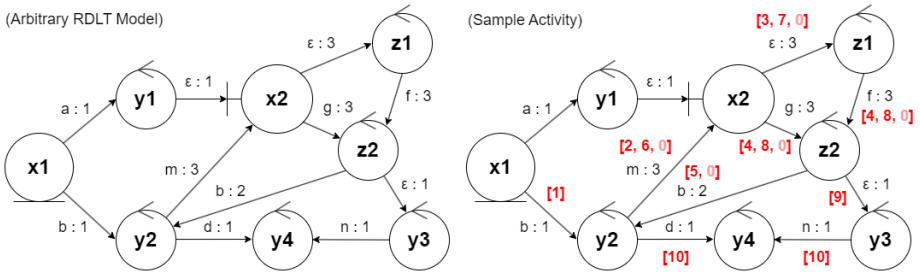

Fig. 2. Sample RDLT Model R and an activity $S = \{S(1), S(2), S(3), S(4), S(5), S(6), S(7), S(8), S(9), S(10)\}$ where $S(1) = \{(x1, y2)\}$; $S(2) = \{(y2, x2)\}$; $S(3) = \{(x2, z1)\}$; $S(4) = \{(x2, z2), (z1, z2)\}$; $S(5) = \{(z2, y2)\}$; $S(6) = \{(y2, x2)\}$; $S(7) = \{(x2, z1)\}$; $S(8) = \{(x2, z2), (z1, z2)\}$; $S(9) = \{(z2, y3)\}$; $S(10) = \{(y2, y4), (y3, y4)\}$

must be satisfied. The limit of an arc, on the other hand, is its maximum allowed number of traversals in one activity extraction. If the node has multiple outgoing arcs, $\mathcal{A}$ chooses one of these arcs nondeterministically, then checks whether the chosen arc can be traversed as described previously. The algorithm $\mathcal{A}$ outputs an **activity profile** [1] that contains the set of components and the execution requirements carried out by the system represented by R. An activity S with a profile $S = \{S(1), S(2), ..., S(k)\}$, $k \in \mathbb{N}$, is a set of sets $S(t)$, where each $S(t)$ contains the set of arcs that are traversed at time step t. This is depicted in Figure 2 as positive integer values in brackets, of the form $[t_1, t_2, ..., t_l]$, where l corresponds to the arc limit. Figure 2 includes only one possible activity profile that can be extracted from the illustrated model.

If an activity has a repeating structure, then there can be multiple activities sharing the same set of arcs, differing on the number of iterations of the repeating structure. A collection of such activities is called an **activity group** [6]. An activity group that is not a subset of some larger activity group is called a **maximal activity group** [6] from which maximal activities can be identified. A **maximal activity** is roughly described in [7] as having an activity profile whose set of traversed arcs is not contained in a larger activity profile. In the context of maximal activity groups, it is the activity that exhausted the maximum number of iterations on the repeating structure. The activity profile in Figure 2 is therefore not a maximal activity. Rather, the activity that traverses the arc $(z2, y2)$ twice would be a maximal activity. Figure 3 shows all maximal activities derivable from the RDLT in Figure 2.

The difference between activities A and B is the traversal of arcs $(x1, y1)$ and $(y1, x2)$ or the lack thereof. Based on the constraints of the arcs, certain control on the flow of graph traversal is imposed. The merge point of two incoming arcs at vertex $x2$ imposes a particular flow control that caused the deviation between activities A and B. This is called a **control structure** in RDLT models [1].

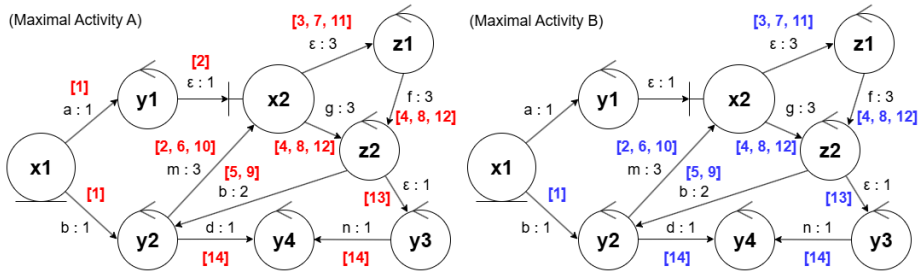

Fig. 3. Maximal Activities of RDLT in Figure 2

2.3 Join and Split Structures in RDLT

The most common control structures used in RDLT modeling are the join and split structures defined in [1,8,9]. In Figure 2, the arcs $(z1, z2)$ and $(x2, z2)$ merge at $z2$ where both arcs have unequal constraints. This implies that both arcs must be traversed in order to satisfy both constraints to proceed to $y3$. Hence, the merge point at $z2$ is called an **AND-join** [1,9]. Note that the merge in $y4$ is also an AND-join. Meanwhile, the arcs $(x1, y2)$ and $(z2, y2)$ have equal constraints. This means that after traversing either one of them, the constraint has been satisfied for both of them, allowing us to proceed to $x2$ without having to wait for the other arc to be traversed. Thus, this merge point at $y2$ is called an **OR-join** [1,9]. If all incoming arcs to a node have no constraints (i.e. $C = \epsilon$), this node also depicts an OR-join. For the case of node $x2$ where one arc has a constraint while the other has none, it behaves differently depending on which arc is traversed first. If $(y2, x2)$ is traversed first, there is no need to traverse $(y1, x2)$ to proceed from $x2$ since the latter has no constraints, thus acting like an OR-join. But if it were the other way around, $x2$ would behave like an AND-join since we cannot proceed from $x2$ unless all incoming arcs with constraints are traversed. Hence, this merge point is called a **MIX-join** [1,9].

If a node has multiple outgoing arcs towards distinct other nodes, this node is called a split. If all such arcs have equal constraints (or all have none), this is called an **AND-split** [1,8]. On the other hand, if all such arcs have varying constraints, it is called an **OR-split** [1,8]. It should be noted that in OR-splits, the execution of one does not imply the nonexecution of the other [1]. They simply imply that at least two process flows can be traversed from the split.

Remark 1. The activity extraction algorithm for RDLTs chooses the next node from some node x to any of its adjacent nodes y in a nondeterministic manner [1]. Hence, OR-splits and AND-splits do not always behave as expected at the point of the split in an RDLT. Only with the use of an eventual join can the said splits be designed according to the aforementioned joins, where the paths of each branch of the split would merge [9].

These structures influence the traversal and extraction of activities in an RDLT model R. In [1], the algorithm $\mathcal{A}$ extracts an activity from R per execution using a depth-first search approach. Another study proposed new algorithms that use a breadth-first search approach to extract multiple activities from R that run in parallel, where Remark 1 is reflected. The next subsection discusses parallel activities in RDLT.

2.4 Parallel Activities in RDLT

The activity extraction algorithm only produces one activity profile from an input RDLT R at a time. The number of traversals was tracked per arc throughout the extraction process, and if it encountered an arc whose traversal limit is reached and it has nowhere else to go to reach the sink, this activity will not complete. Consider having identified two distinct activities S and S' of R. If there is a set of arcs that appeared in both activities, these arcs are called the **shared resource** between S and S' [2]. If we try to run both activities in R at the same time, it is likely that their shared resource would reach their traversal limits earlier. This introduces the risk that one of the activities of R may not be completed if we try to run them in parallel. We call S and S' **competing processes** when $S(S')$ cannot be completed in R if $S'(S)$ is allowed to complete [2]. Taking as an example the activities A and B in Figure 3, the arcs traversed that are common between them are the entire set of arcs used in activity B. Since both activities use $(x1, y2)$ at time step 1 and $L((x1, y2)) = 1$, only one of them can complete, and thus A and B are competing processes. To avoid this, we can reconfigure the limits of the shared resources between activities A and B to allow both to complete. Figure 4 shows this modified version of R with both activities A and B completed at time step 14. By Definition 1, this means that A and B are **parallel activities** in R.

Definition 1. ***Parallel Activities for Simple RDLT*** *(based on [2])*
A set of maximal activities A of an RDLT R are parallel if for every pair of activities $S, S' \in A$, the following conditions hold:

1. *S and S' have the same set of input and output vertices;*
2. *S and S' do not compete with each other; and*
3. *S and S' complete at the same time.*

In order to determine parallel activities in an RDLT R, two algorithms are proposed in [2], namely the **Generation of Traversal Tree** (GTT) and **Parallel Activity Extraction** (PAE) algorithms. The former generates all possible maximal activities in R into a so-called *traversal tree*, then identifies those that run in parallel, while the latter outputs the activity profiles of the parallel maximal activities in R. Both algorithms follow the rules for producing separate and merged activity sets established in [2].

In summary, the technique used in both algorithms is to divide the time points into time slices running from source to sink and determine whether at a certain time slice, the multiple maximal activities in R are merged or separate.

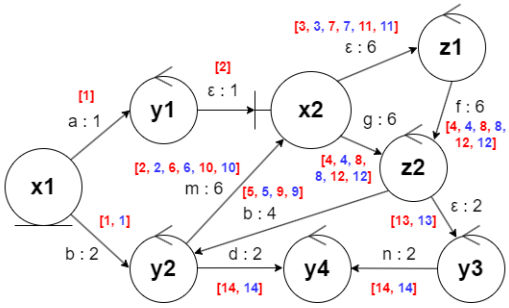

Fig. 4. Parallel Maximal Activities of the Modified RDLT

To determine the time slices, we consider the splits and joins in RDLT (see Section 2.3). The scenarios per structure can be roughly described as follows:

1. **OR/AND-split at vertex** x - The boundary where activities that are initially merged become separate activities across each outgoing arc from x.
2. **MIX/AND-join at vertex** y - For this case of MIX-join, some constraints are unsatisfied. This is the boundary where multiple activities that are initially separate across all incoming arcs to x become merged into one activity.
3. **MIX/OR-join at vertex** y - For this case of MIX-join, no constraints are left unsatisfied. The activities reaching y continue to be separated and proceed independently of each other.

Scenario 1 always induces the separation of activities regardless of the split type. Whether these activities remain separated or eventually merge depends on the type of eventual joins, thus supporting the discussion in Remark 1. At the end of the algorithms, the activities that remained separate will be the set of maximal activities of R from which those that run in parallel are identified. The enrichment of RDLT in [3] includes a new split type that enforces mutually exclusive paths. This is likely to have an impact on the current implementation of extracting parallel activities, as well as on its related concepts. We discuss more about this enrichment in the next subsection.

2.5 Enriched Robustness Diagram with Loop and Time Controls

In [3,4], a variant of RDLT is established to address its identified limitations called the *Enriched Robustness Diagram with Loop and Time Controls* (ERDLT). Figure 5 shows an arbitrary ERDLT model and its set of components. It has additional vertex and arc types that can better capture object-oriented programming concepts in workflow or system modeling. Real-world systems and use case texts were modeled using ERDLT in [3,4] and demonstrated how system representation is improved compared to RDLT.

New vertex types include **entity(boundary) proxy objects** which serve as instances of entity(boundary) objects, **guards** that represent decision con-

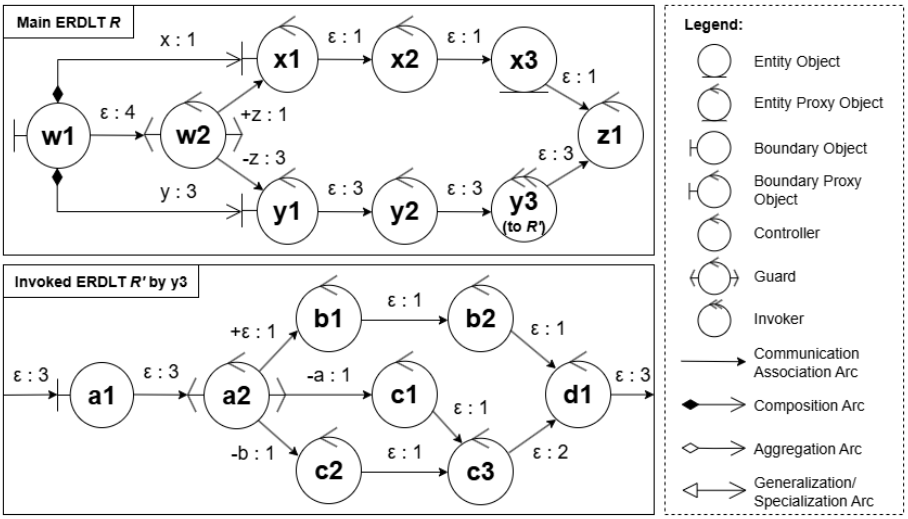

Fig. 5. Sample ERDLT Model and its Components

trol objects and mutually exclusive paths, and **invokers** that represent abstracted subprocesses or invoke another layer of system processes [3]. Meanwhile, new arc types include **composition**, **aggregation**, and **generalization/specialization** arcs, which represent their respective hierarchical abstractions or relationships with multiple layers of entities [10,11].

To map an instantiation of an object, an entity(boundary) proxy object y can be connected by a regular arc from an entity(boundary) object x. The arc (x, y) is, therefore, called an **instantiation dependency** [3]. Moreover, an **XOR-split** [3] is also defined with guard vertices. This type of split enforces mutually exclusive paths, i.e., the traversal to one path implies the non-traversal of the other. In Figure 5, the vertices $w2$ and $a2$ are guards whose outgoing arcs have at least one positive-signed and one negative-signed C-value. The former(latter) represents the path to traverse if the condition of the guard is satisfied(unsatisfied), then blocks the latter(former) path.

These additional vertex and arc types have introduced more structures or control flow designs can be derived in ERDLT models, including XOR-splits that cannot be strictly imposed in RDLT. This implies that the previously defined structures, behavior, and properties in RDLT models may not be exactly applicable to ERDLT models, including the concept of parallel activities.

3 Problem Synthesis and Research Goals

The new structures and behavior introduced by the new sets of vertices and arcs in ERDLT could imply that the defined concepts to establish parallel activities in

RDLT may not be fully applicable to ERDLT. Hence, there is a need to determine how these concepts would be defined in ERDLT models. This is to extend the improvement in system modeling offered by ERDLTs to parallel workflows and to allow the representation, analysis, and verification of such workflows in ERDLT. In this study, we focus on the structural aspects of parallel activities in an ERDLT R, including an algorithm to generate traversal trees for R.

4 Methodology

In this section, the definitions provided in Section 2 will be evaluated to determine whether they can be fully reused in the context of ERDLT models, and if not, modifications will be proposed. In particular, the concepts of activity groups, maximal activities, and the rules to produce separate and merged activity sets (PSMAS) will be defined under ERDLT. The other definitions will also be identified for their reusability under ERDLT. The Generation of Traversal Tree (GTT) algorithm will also be modified to take into account the new vertices introduced in ERDLT, and lastly, a design in implementing the modified GTT in parallel code will be suggested.

4.1 Concepts of Parallel Activities in ERDLT

Table 1 roughly summarizes the concepts defined in Sections 2.2 and 2.4, and whether they can be fully reused in the context of ERDLT models and what considerations or additions to be made. We focus only on the structural aspect, hence only the concepts of *activity groups*, *maximal activities*, *PSMAS rules*, and the *algorithm for generating traversal trees* will be established for ERDLT.

No.	**Relevant Concepts** [2,6,7]	**Reusability to ERDLT** [3]
1	Activity Groups	Fully reusable
2	Maximal Activities	Fully reusable, dependent to item 1
3	Shared Resource	Fully reusable
4	Competing Processes	Fully reusable, dependent to item 3
5	PSMAS Rules	Needs additional rules for XOR-splits
6	Parallel Activities	Fully reusable, dependent to items 2, 4, and 5

Table 1. Relevant Concepts to Parallel Activities in ERDLT

An activity group in an RDLT R is a group of distinct activity profiles with the same set of vertices traversed. They only vary in the number of iterations on the repeating structure in the said activity profiles. Even with the new vertex and arc types in ERDLT, this definition still applies. Consequently, the definition of maximal activities for RDLT also applies to ERDLT. However, ERDLT models can now represent mutually exclusive paths (XOR-splits), which implies that the PSMAS rules need to add cases for this structure. This type of split is induced by guard objects, but not all splits they induce are XOR-splits according to its definition in [12]. The next subsection discusses more about these splits.

4.2 Split Types from Guard Vertices in ERDLT

The guard vertex was established in [3] with the purpose of having an explicit representation of mutually exclusive paths or XOR-splits. An XOR-split is defined as a point in the workflow process where only one of several branches is chosen after a decision or some workflow control data [12]. However, the design of guard vertices allows multiple branches to be chosen. Guards can have more than one positive-signed or negative-signed outgoing arc; thus, even after blocking all positive-signed outgoing arcs when the guard condition was not satisfied, more than one negative-signed outgoing arc can be traversed. Hence, we introduce the different types of split that can be induced by guard vertices.

We first introduce new terms with respect to the types of outgoing arc from a guard vertex. From this point on, positive-signed outgoing arcs will generally be referred to as **true arcs**, while negative-signed outgoing arcs will be referred to as **false arcs**. This is because the former(latter) are the arcs that will be left available for traversal when the guard condition is met(unmet) while the latter(former) will be blocked by maxing out their traversal limits. The types of split at guard vertices will be distinguished by the number of true and/or false arcs. Moreover, we refer to the blocked arcs as **non-satisfying arcs**, while the arcs left available for traversal are referred to as **satisfying arcs**. Using a modified sign function $sgn_E()$ that takes arcs $(x, y) \in E$ as input, which returns 1 if (x, y) is a true arc and -1 if it is a false arc, we define the following splits:

1. **(Strictly) XOR-split** - There is exactly one true arc and one false arc from x, where $\exists y, z \in V, \exists (x, y), (x, z) \in E$, and $sgn_E(C((x, y))) = \neg sgn_E(C((x, z)))$.
2. **Partial XOR-split** - There is exactly one true/false arc and the opposite is more than one from x, where $\exists y \in V$, $\exists Z \subset V$ and $y \notin Z$, $\exists (x, y) \in E$ and $\exists (x, z_i) \in E$ for every $z_i \in Z$, $i \geq 2$, such that $sgn_E(C((x, y))) = \neg sgn_E(C((x, z_i)))$.
3. **Pseudo XOR-split** - There are more than one true and false arcs from x, where $\exists Y, Z \subset V$ and $Y \cap Z = \emptyset$, and $\exists (x, y_i), (x, z_j) \in E$ for every $y_i \in Y$, $z_j \in Z$, $i \geq 2$, $j \geq 2$, such that $sgn_E(C((x, y_i))) = \neg sgn_E(C((x, z_j)))$.

Figure 6 shows an example of the split types enumerated that a guard vertex can induce. Each type has an impact on the additional rules for producing separate and merged activity sets and on the generation of traversal trees of ERDLT models that contain guard vertices (see Appendix A for full formal definition).

4.3 Rules in Producing Separate Activity Sets on Guard Vertices

Since all split and join structures in RDLT are fully acquired by ERDLT [3,4], the existing rules for producing separate and merged activity sets in [2] will also apply in ERDLT (see Section 2.4). In addition to the three cases of this rule, we will only add cases to handle split types from guard vertices.

4. **Strictly/Partial XOR-split at vertex** x - For this case of partial XOR-split, there is only one outgoing arc allowed for traversal from x upon evaluating whether its condition is met or not. The guard x proceeds to block arcs

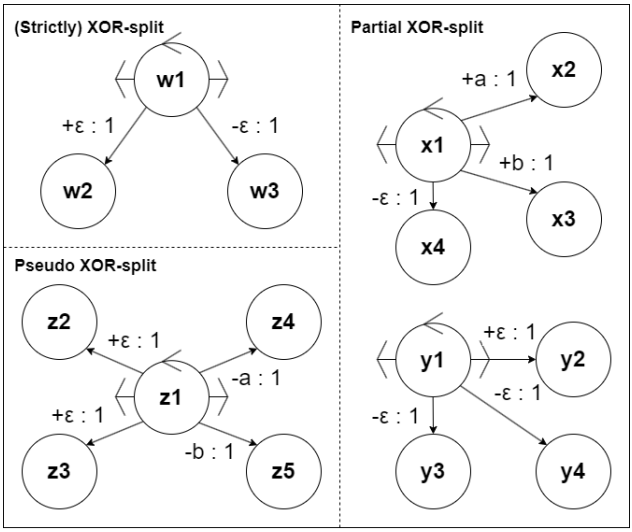

Fig. 6. Split Types on Guard Vertices

of a certain sign (positive or negative), and no separate profiles are induced; hence, the process flow(s) of the existing profile(s) is(are) retained.

5. **Pseudo/Partial XOR-split at vertex** x - For this case of partial XOR-split, there are more than one outgoing arcs allowed for traversal from x upon evaluating whether its condition is met or not. The guard x proceeds to block arcs of a certain sign (positive or negative), and a separate activity profile S' for each remaining arc (x, y_i) in the ERDLT, S' adopts the set of arcs traversed in S built from source to x.

4.4 Proposed Generation of Traversal Tree Algorithm for ERDLT

To generate traversal trees in ERDLT models, we propose a modification of the GTT algorithm from [2] such that the new types of vertices and arcs will be taken into account. The entity and boundary proxy objects can be treated similarly to the other vertex types, while an invoker y can be treated as bridges between ERDLT models $R = \{V, E, T, M\}$ and $R' = \{V', E', T', M'\}$, where if $\exists(x, y) \in E$ for some $x \in V$ and $\exists z \in V'$ such that z is pointed to by an **invoking arc** in E', it can be interpreted as if the vertices x, y, and z are directly connected by arcs (x, y) and (y, z). Similarly for $\exists u \in V'$ that has an outgoing arc that is a **returning arc** in E', $\exists v \in V$, and $\exists(y, v) \in E$, it can be viewed as if u, y, and v are directly connected by (v, y) and (y, v). For the new relationship types in ERDLT models, they mainly represent object-oriented relationships between models and still depict process flows. Such relationships or arc types on their own do not necessarily impose any kind of restriction on their traversal aside from the process flow that they explicitly represent. Therefore, this modification

to GTT is specifically aimed at dealing with guard objects which induce special types of splits (see Algorithm 3 in Appendix C for the full pseudocode).

Similarly to the original version, the method `Check`(x, y_j) (see Algorithm 2 in Appendix B) checks whether a vertex x has any adjacent vertex y_j, $j = 1, 2, ...$ where every $\exists(x, y_j) \in E$ is given a branch in the traversal tree. The difference in Algorithm 3 is the additional lines $14 - 20$, $23 - 25$, $58 - 62$, and $65 - 67$ that handle invokers and guards. The former shifts the branch generation on the same traversal tree between an invoked ERDLT and its caller ERDLT, while the latter calls the method `CheckGuardCondition`(x) (see Algorithm 4 in Appendix C) which is fully reused from the modified activity extraction algorithm $\mathcal{A}'$ for ERDLT models. Upon execution of the blocking of non-satisfying arcs, Algorithm 3 will not be able to generate branches for those arcs in the traversal tree. This implies that in an ERDLT R that has only one guard vertex, R has two possible traversal trees: one for when the guard condition is satisfied and another for when it is not satisfied. Hence, the number of possible traversal trees that can be generated from R grows along with the number of guard vertices in R.

Using Algorithm 3, the ERDLT model in Figure 5 has a total of three distinct traversal trees illustrated in Figure 7 below. Due to the blocking mechanism of the guard vertices, some vertices and arcs in an EDRLT R will not be present in its generated traversal tree, unlike those of RDLT models. Therefore, there will be multiple distinct traversal trees in ERDLT models depending on whether the conditions of its guard vertex are satisfied or not, or the combinations of that if there are multiple guard vertices. Such combinations can be observed in the trees Tr^{-+} and Tr^{--} in Figure 7 (the superscripts are the sequence of satisfying arcs per guard) when the condition of guard $w2$ in Figure 5 is not met. Theorems 1, 2, and 3 demonstrate the consistency, correctness, and asymptotic complexity of Algorithm 3, respectively. See Appendix D for their respective proofs.

Theorem 1. *Given an ERDLT R and an RDLT R' where R and R' are isomorphic to each other and have equal attribute values, Algorithm 3: MGTT generates its traversal tree from R that is equivalent to the traversal tree generated by Algorithm 1: GTT from using R'.*

Theorem 2. *Given an ERDLT R, a maximal activity S can be derived from R using the activity extraction algorithm $\mathcal{A}'$ in [4] if and only if there exists a branch in the set of traversal trees that corresponds to the reachability sets (or sets of arcs) of S, that is, at time step i, both the activity extraction algorithm and Algorithm 3: MGTT traverse an arc and a branch, respectively.*

Theorem 3. *The space and time complexity of Algorithm 3: MGTT is $O(2^n)$, where $n = |V| + |E|$, V and E are the sets of vertices and edges in an ERDLT.*

4.5 Design Suggestions on the GTT Algorithm in Parallel Code

We now present some suggestions or considerations if we try to design a version of the GTT algorithm if it were to be implemented in CUDA [13,14]. The CUDA

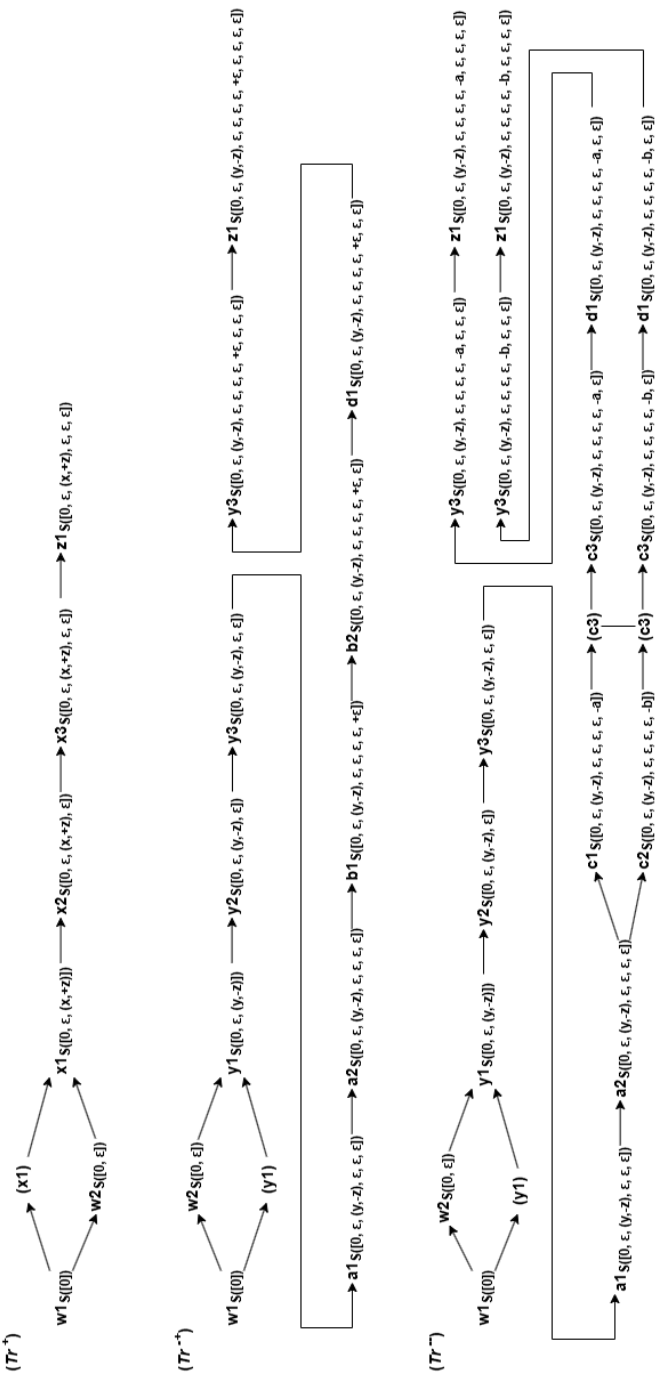

Fig. 7. Three Possible Traversal Trees of R in Figure 5

programming model is basically an integration of host and device code, where the former is the sequential parts of the code executed in CPU while the latter is the highly parallel parts that are executed on the GPU and usually includes proper allocation and utilization of threads, thread blocks, and grid of blocks. If we examine parts of GTT (both original and modified versions), there are two `while` loops that are not nested. The first is for dealing with maximal activities with iterations, while the second is for those without iterations [2]. We first focus on the case where an RDLT model R (hence, the original version of GTT) has no iterations such as R in Figure 2 without $(z2, y2)$ (called a looping arc [1]).

The original generation of traversal tree algorithm takes a classical sound (meaning that all vertices and arcs are reachable [9]) RDLT R with one source and one sink as input. R includes its sets of vertices V and arcs E. One way of mapping these into CUDA programming is that the entire R will be processed in a grid with only one block, where one thread per arc in E will be allocated to process the generation of their branches. Then, we retrieve the output per branch and divide them into small groups. Threads will then be allocated per group, then merging of the branches will be performed on every group if there are common vertices between them. We do this until all branches are connected into one traversal tree of R. This implies a better time complexity than that of Algorithm 3, possibly $O(n \log n)$ if the assigning of groups is designed properly and accordingly. Figure 8 illustrates this flow of traversal tree generation in parallel in the RDLT model in Figure 2 excluding the looping arc $(z2, y2)$.

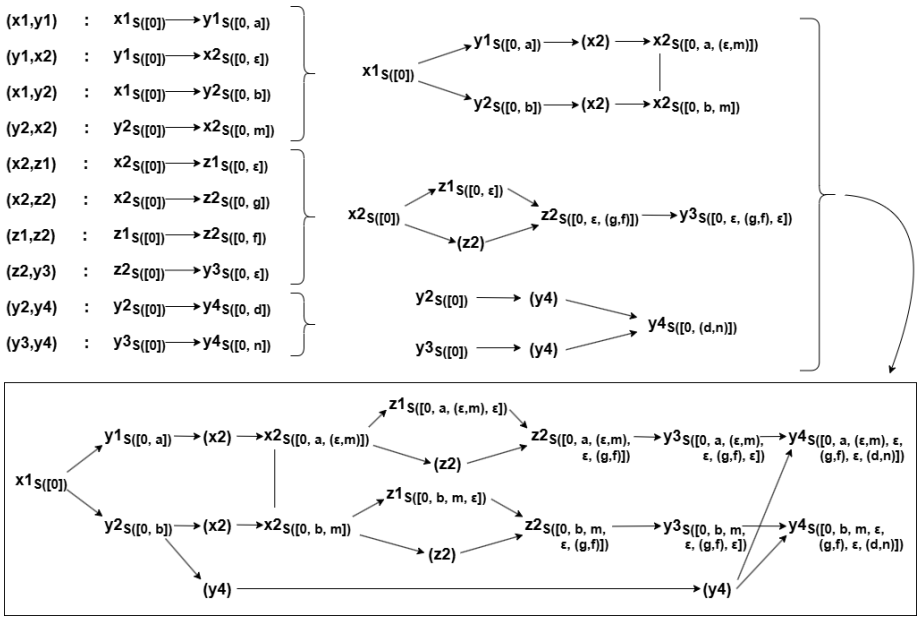

Fig. 8. General Flow of Traversal Tree Generation per Branch in Parallel

This general flow only needs to be able to handle iterations, and the parallel code version of GTT for RDLT models will theoretically be complete. If we consider the blocking of guard objects to non-satisfying arcs, hence the modified GTT for ERDLT models, we can utilize more blocks in a grid for generating its traversal tree. Taking, for example, the ERDLT model R in Figure 5 which has three distinct traversal trees illustrated in Figure 7, we can allocate three blocks to generate these trees separately and concurrently. Each block will only need some indicators of the set of satisfied and/or blocked arcs by the guard vertices. However, determining the maximum number of distinct traversal trees from the ERDLT models will introduce more complexity into the code.

5 Results and Discussion

The quirk of guard vertices covered most of the discussion in this paper. It was originally established to allow ERDLT in modeling or representing mutually exclusive paths or XOR-splits. However, based on the formal definition of guards and the technical definition of XOR-splits, it was found that guard vertices do not necessarily equate to the presence of (strictly) XOR-splits. In summary, we defined *true* and *false arcs*, *satisfying* and *non-satisfying arcs*, and the additional split types of guards, namely the *partial* and *pseudo XOR-splits* (see Section 4.2). Moreover, we proposed an algorithm to generate traversal trees from ERDLT models in which the behavior of guards is implemented (see Algorithm 3 in Appendix C). Due to the blocking mechanisms of the guard objects, not all maximal activities of an ERDLT model R can be seen in one generated traversal tree if R contains at least one guard object. This implies that guard objects have a significant impact on extracting parallel activities in ERDLT, such that the pairs of maximal activities that do not belong in the same traversal tree will never be parallel activities. This is because the activities that use the satisfying arcs of a guard vertex will never execute simultaneously with those that use the non-satisfying arcs. Therefore, we present the following theorem (whose proof is also in Appendix D):

Theorem 4. *For the set of traversal trees of an ERDLT model R generated by Algorithm 3: MGTT, the parallel activities of R can be found only among the maximal activities in the same traversal tree.*

For an ERDLT model R, its set of maximal activities is grouped into those that use the satisfying arcs of a guard vertex, and those that use the non-satisfying arcs in separate traversal trees. If there are multiple guards, combinations of whether or not their conditions were met in activity profiles should also be considered. In Figure 7, the trees Tr^{+} and Tr^{-+} show one maximal activity, respectively, while Tr^{--} shows two maximal activities. With Theorem 4, we can only determine the maximal activities that could be parallel among those shown in Tr^{--} (since the other trees each contain only one maximal activity). All conditions for maximal activities to be parallel (see Definition 1) can be cross-checked between R in Figure 5 and Tr^{--} in Figure 7. The activities S

and S' in Tr^{--} have the same input and output vertices and both completed at time step 11. Moreover, the L-attribute values of their shared resources can accommodate the execution of both S and S' at the same time. Therefore, S and S' are parallel activities in R.

The work presented in this study only covers the structural aspects of parallel activities in ERDLT models. In addition to the identified reusability mapping of the concepts for these activities in Table 1, the results of this study can be used to address the behavioral aspects of parallel activities in ERDLT. Furthermore, such results can help in the formalization and verification of the soundness property of ERDLT models, particularly the notions of soundness that consider multiple activities running concurrently. This is because the literature that formalized weak and easy soundness in RDLT [15] has noted that there will be no distinction between easy and lazy soundness in RDLT models unless multiple concurrent activities are considered. This has been explored in [16] using the established concepts, results, and evidence in [2,15] and has succeeded in formalizing lazy soundness for RDLT models with concurrent activities. Hence, our work can serve as a preliminary step in the formalization of such notions of soundness in ERDLT models in addition to extending ERDLT to modeling parallel workflows.

6 Conclusions and Future Work

The improved level of representation that ERDLT offers compared to RDLT has its strength in integrating object-oriented concepts in workflow modeling. Its usefulness can be extended to parallel workflows by formalizing parallel activities in ERDLT. Our study provides the first step towards achieving this goal. The definitions of relevant concepts in establishing parallel activities in RDLT were identified and an overview of their reusability was provided in the context of ERDLT models. This mapping includes both structural and behavioral aspects of parallel activities in ERDLT, but this paper is focused on the structural aspect where a modification to the algorithm for generating traversal trees is proposed to handle the new vertex types, specifically the guard vertices. This is due to the significant impact of such vertices on activity extractions that are reflected in having multiple traversal trees of the same ERDLT model. We found that parallel maximal activities of ERDLTs can only be determined among those grouped in the same traversal tree.

Suggestions for algorithm design of a parallel code version for the generation of traversal trees were also provided and discussed. This study mainly focused on theoretical results, and thus we suggest the implementation and validation of the presented design in future work, as well as to cover the behavioral aspects of parallel activities in ERDLT models and to determine structural profiles that have implications on the notions of soundness in such models.

Acknowledgment. E. Petilos would like to thank the Engineering Research and Development for Technology scholarship program of the Department of Science and Technology - Science Education Institute in the Philippines for funding his graduate studies at

the University of the Philippines Diliman during which the work presented in this study was conceptualized. R.A. Juayong acknowledges Robert Cheng, the URATEX Professional Chair in Engineering. F.G. Cabarle thanks the Scientific Productivity System of the University of the Philippines (2021-2023) for their support.

References

1. Malinao, J.: On building multidimensional workflow models for complex systems modelling. PhD thesis, Vienna University of Technology (2017). doi.org/10.34726/hss.2017.43523
2. Doñoz, R.: Parallel activities in robustness diagram with loop and time controls. Undergraduate Thesis, University of the Philippines Tacloban College (2024).
3. Petilos, E., Malinao, J.: Enriched robustness diagram with loop and time controls for scalable model representation. In: Krouska, A., Mylonas, P., Caro, J. (eds) Novel and Intelligent Digital Systems: Proceedings of the 5th International Conference (NiDS 2025). Lecture Notes in Networks and Systems, vol 1707, 64-87. Springer, Cham. (2026) doi.org/10.1007/978-3-032-10824-1_6
4. Petilos, E., Malinao, J.: Building scalable hierarchical abstractions in the enriched robustness diagram with loop and time controls. Discover Computing, 28(1), 1-33 (2025). doi.org/10.1007/s10791-025-09649-4
5. Rosenberg, D., Stephens, M.: Use case driven object modeling with UML: theory and practice. Reading: Addison-Wesley Professional (1999).
6. Eclipse, K., Malinao, J.: Model decomposition of robustness diagram with loop and time controls to sequence diagrams using activity groups. Pre-proceedings of the Workshop on Computation: Theory and Practice (WCTP), Hokkaido, Japan (2023).
7. Malinao, J., Juayong, R.A.: Model separability of robustness diagram with loop and time controls. Science & Engineering Journal, Philippine-American Academy of Science & Engineering (2024). doi.org/10.54645/2024172CRV-31
8. Roca, T.N.: Well-handledness in robustness diagram with loop and time controls. Undergraduate Thesis, University of the Philippines Tacloban College (2024).
9. Malinao, J., Juayong, R.A.: Classical soundness in robustness diagram with loop and time controls. Philippine Journal of Science, 152 (2023). doi.org/10.56899/152.6B.06
10. Rumbaugh, J., Jacobson, I., Booch, G.: The unified modeling language reference manual second edition. Pearson Higher Education (2004). ISBN:0-321-24562-8
11. OMG UML: Object Management Group Unified Modeling Language Version 2.5 (2012)
12. Van Der Aalst, W. M., Ter Hofstede, A. H., Kiepuszewski, B.: Workflow patterns. Distributed and parallel databases, 14, 5-51 (2003). doi.org/10.1023/A:1022883727209
13. NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html
14. Kirk, D.B., Hwu, W.W.: Programming massively parallel processors: a hands-on approach. Morgan kaufmann, p. 156 (2016).
15. Ramirez, R.I.: On weak, lazy, and easy soundness in robustness diagram with loop and time controls. Undergraduate Thesis, University of the Philippines Tacloban College (2024).
16. Afable, M.E., Malinao, J.: On lazy soundness of robustness diagram with loop and time controls. Accepted for presentation at the Workshop on Computation: Theory and Practice (WCTP 2025), December 1-3, 2025, Hokkaido, Japan. To appear in Post-Proceedings of the WCTP 2025 by Atlantis Press.

A Formal Definition of ERDLT

Definition 2. ***Enriched Robustness Diagram with Loop and Time Controls (ERDLT)*** *[4]*
An ERDLT is a graph representation R of a system that is defined as $R = (V, E, T, M)$ where

- *V is a finite set of vertices where every vertex has a type $V_{type} : V \rightarrow \{$'b', 'bp', 'e', 'ep', 'c', 'g', 'i'$\}$, where 'b', 'bp', 'e', 'ep', 'c', 'g', and 'i' means the vertex is either a "boundary object", "boundary proxy object", "entity object", "entity proxy object", "controller", "guard", or an "invoker", respectively. Furthermore, $V = V_o \cup V_p \cup V_c$ where:*
  - *$V_{type} : V_o \rightarrow \{$'b', 'e'$\}$*
  - *$V_{type} : V_p \rightarrow \{$'bp', 'ep'$\}$*
  - *$V_{type} : V_c \rightarrow \{$'c', 'g', 'i'$\}$*

  These subsets will be generally called as "object group", "proxy group", and "controller group", respectively.
- *E is a finite set of arcs where every arc has a type $E_{type} : E \rightarrow \{$'ca', 'a', 'c', 'g'$\}$, where 'ca', 'a', 'c', and 'g' means the arc is either a "communication association", "aggregation", "composition", or a "generalization", respectively. Furthermore, $E = E_{ca} \cup E_{ha}$ where:*
  - *$E_{type} : E_{ca} \rightarrow \{$'ca'$\}$, $E_{ca} \subseteq (V \times V) \backslash E'_{ca}$ where $E'_{ca} = \{(x, y) | [x, y \in V_o] \vee [x, y \in V_p$ where $V_{type}(x) \neq V_{type}(y)] \vee [V_{type}(x) \in V_p$ and $V_{type}(y) \in V_o]\}$*
  - *$E_{type} : E_{ha} \rightarrow \{$'a', 'c', 'g'$\}$, $E_{ha} \subseteq [(V_o \cup V_p) \times (V_o \cup V_p)] \backslash E'_{ha}$ where $E'_{ha} = \{(x, y) | [x, y \in \{$'e', 'ep'$\}$ where $V_{type}(x) \neq V_{type}(y)] \vee [V_{type}(x) =$ 'bp' and $V_{type}(y) =$ 'b'$] \vee [V_{type}(x) =$ 'b' and $V_{type}(y) =$ 'bp' where $E_{type}((x, y)) =$ 'g'$] \vee [x, y \in V_p$ where $V_{type}(x) = V_{type}(y)$ and $E_{type}((x, y)) =$ 'g'$]$*

  These subsets will be generally called as "communication association group", and "hierarchical abstraction group", respectively, with the following attributes whose values are derived from the following functions,
  - *$C : E \rightarrow \Sigma \cup \{\epsilon\}$ where Σ is a finite non-empty set of symbols and ϵ is the empty string. Note that for real-world systems, a task $v \in V$, i.e. $V_{type}(v) \in V_c$, is executed by a component $u \in V$, $V_{type}(u) \in V_o \cup V_p$, or an instance $w \in V$, $V_{type}(w) \in V_p$ is created from an object $o \in V$, $V_{type}(o) \in V_o \cup V_p$. This component-task association and instantiation relationship are represented by the communication association arcs $(u, v), (o, w) \in E$, $E_{type}((u, v)) = E_{type}((o, w)) =$ 'ca'. Additionally, components i can be parents of other components j only when $i, j \in V_o \cup V_p$ where $V_{type}(i) = V_{type}(j)$ or if $V_{type}(i) =$ 'b' and $V_{type}(j) =$ 'bp' where $E_{type}((i, j)) \neq$ 'g'. This object-to-object relationship is represented by hierarchical abstraction arcs $(i, j) \in E$, $E_{type}((i, j)) \in E_{ha}$. Moreover, $C((x, y)) \in \Sigma$ represents a constraint to be satisfied to reach y from x. This constraint can represent either an input requirement or a parameter $C((x, y))$ which needs to be satisfied to proceed from using the*

component/task x to y. $C((x,y)) = \epsilon$ represents a constraint-free process flow to reach y from x or a self-loop when $x = y$. Furthermore, outgoing arcs $(m,n) \in E$ of a guard $m \in V$ to any other vertex $n \in V$ must have ***signed C-values****, hence, $C((m,n)) \in C_p \cup C_n$ where:*

- *— $C_{type} : C_p \rightarrow +\Sigma \cup \{+\epsilon\}$ where arcs (m,n) with such C-value will be blocked for traversal if the condition of the guard is not satisfied. Otherwise, these arcs can be traversed.*
- *— $C_{type} : C_n \rightarrow -\Sigma \cup \{-\epsilon\}$ where arcs (m,n) with such C-value will be blocked for traversal if the condition of the guard has been satisfied. Otherwise, these arcs can be traversed.*

- *$L : E \rightarrow \mathbb{Z}^+$ is the maximum number of traversals allowed on the arc.*

– *Let T be a mapping such that $T((x,y)) = (t_1, \ldots, t_n)$ for every $(x,y) \in E$ where $n = L((x,y))$ and $t_i \in \mathbb{N}$ is the time a check or traversal is done on (x,y) by some algorithm's walk on R.*

– *$M : V \rightarrow \{0,1\}$ indicates whether $u \in V$ and every $v \in V$ where $(u,v) \in E$ and $C((u,v)) = \epsilon$ induce a subgraph G_u of R known as a* ***reset-bound subsystem****(RBS). The RBS G_u is induced with the said vertices when $M(u) = 1$. In this case, u is referred to as the* ***center*** *of the RBS G_u. G_u's vertex set V_{G_u} contains u and every such v which includes (a) the child objects owned by u, if any, (b) all vertices owned by u and by its child objects, (c) instances v of u, and (d) its arc set E_{G_u} has $(x,y) \in E$ if $x, y \in V_{G_u}$.*

- *If the center u is a parent in a composition of another vertex v, hence $\exists (u,v) \in E$ where $E_{type}((u,v)) =$ 'c', where $V_{type}(u) =$ 'b', $V_{type}(v) =$ 'bp', and then $\exists w \in V$ where $V_{type}(w) =$ 'i' and $(v,w) \in E$ where $E_{type}((v,w)) =$ 'ca', then G_u is a* ***nested RBS*** *where the invoked RDLT R' of w has a center $o \in V$ where $V_{type}(o) =$ 'b' and $M(o) = 1$, hence an RBS.*
- *An arc $(a,b) \in E_{ca}$ is called an* ***in-bridge*** *of b if $a \notin V_{G_u}$, $b \in V_{G_u}$. Meanwhile, $(b,a) \in E_{ca}$ is called an* ***out-bridge*** *of b if $b \in V_{G_u}$ and $a \notin V_{G_u}$.*
- *For every invoker $c \in V$ where $V_{type}(c) =$ 'i' and R' is the other ERDLT being invoked by c, the arc $(c,d) \in E_{ca}$ of R' is called an* ***invoking arc*** *of R' where the vertex $d \in V'$, $V_{type}(d) =$ 'b' acts as a source vertex of R'. On the other hand, $(d,c) \in E_{ca}$ of R' is called a* ***returning arc*** *to R where the vertex $d \in V'$ acts as a sink vertex of R'.*
- *Lastly, arcs $(a,b), (c,d) \in E$ are* ***type-alike*** *if $\exists y \in V$ where $(a,b), (c,d) \in Bridges(y)$ with $Bridges(y) = \{(r,s) \in E_{ca} | (r,s)$ is either an in-bridge or out-bridge of $y\}$, if $\exists x \in V$ where $(a,b), (c,d) \in Invoke(x)$ with $Invoke(x) = \{(r,s) \in E_{ca}$ of $R \cup E_{ca}$ of $R' | (r,s)$ is either an invoking or returning arc of $x\}$, or if $\forall y \in V$, $(a,b), (c,d) \notin Bridges(y) \cup Invoke(x)$.*

B Generation of Traversal Tree Algorithm for RDLT

Algorithm 1 Generation of Traversal Tree for RDLT [2], $Tr(t)$, $t = 1, 2, ..., k$, $k \in \mathbb{N}$. Assumption per branch: Independent treatment of maximal activities when performing checks and traversals.

Input: RDLT R, with one source vertex s and one sink vertex f
Output: Traversal tree, set of parallel and non-parallel maximal activities

Set $i = 1$
Set $Tr(1) = [s]_{S([0])}$
Set $Ancestors = \emptyset$
while $TRUE$ **do**
  Let X be the set of vertices in Tr with no outgoing branches pointing to ancestral vertices, $x \in X$, $x \neq pending$, i.e. $([x])$
  if $x \in X$, $x = f$ and no other observed continuing branch **then**
    Set $parallelActivities_p$ to be the set of activities that reaches $[f]$ at the same times, $1 \leq p \leq k$, with $eRU(x, y) \in E \geq$ its actual use
    Set $UnparallelActivities_q$ to be the set of activities that reaches $[f]$ at different times, $1 \leq q \leq k$, with $eRU(x, y) \in E \geq$ its actual use
    return $[Tr(t)$, $parallelActivities_p$, $UnparallelActivities_q$, $\forall p, q]$
  else if $X = \emptyset$ **then**
    return 0
  end if
  for each $(x, y_j) \in E$ where $x \in X$ **do**
    if $Check(x, y_j) = TRUE$ and y_j is in $Ancestors$, such that $eRU(x, y_j) \in E \geq$ its actual use **then**
      if (x, y_j) is unconstrained **then**
        Set $prevConstraints = S([0, satisfiedConstraintAncestralArc])$
        Set $updatedConstraints = S([0, satisfiedConstraintAncestralArc, currentSatisfiedArcConstraint])$
        Append $Tr(i)$ a branch $[x]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
      else
        Append $Tr(i)$ a branch $[x] \rightarrow ([y_j])$
        Checks for possibility to resolve $\rightarrow ([y_j])$
        if $\rightarrow ([y_j])$ is an AND or MIX merge point **then**
          Get max time t from $[x_k] \rightarrow ([y_j])$
          Append to $Tr(t)$ the branch $[x_k]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
          if $\rightarrow ([y_j])$ is a MIX or OR merge point **then**
            Append to $Tr(t)$ the branch $[x]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
          end if
        end if
      end if
    end if
  end for
  Append to $Ancestors$ the vertex x
  $i++$
end while
while $TRUE$ **do**

Let X be the set of vertices in Tr with no outgoing branches and $x \in X$, x is not a pending vertex, i.e. $([x])$
if $x \in X$, $x = f$ and no other observed continuing branch **then**
  Set $parallelActivities_p$ to be the set of activities that reaches $[f]$ at the same times, $1 \leq p \leq k$
  Set $UnparallelActivities_q$ to be the set of activities that reaches $[f]$ at different times, $1 \leq q \leq k$
  return $[Tr(t), parallelActivities_p, UnparallelActivities_q, \forall p, q]$
else if $X = \emptyset$ **then**
  return 0
end if
for each $(x, y_j) \in E$ where $x \in X$ **do**
  if $Check(x, y_j) = TRUE$ **then**
    if (x, y_j) is unconstrained **then**
      Set $prevConstraints = S([0, satisfiedConstraintAncestralArc])$
      Set $updatedConstraints = S([0, satisfiedConstraintAncestralArc, currentSatisfiedArcConstraint])$
      Append $Tr(i)$ a branch $[x]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
    else
      Append $Tr(i)$ a branch $[x] \rightarrow ([y_j])$
      Checks for possibility to resolve $\rightarrow ([y_j])$
      if $\rightarrow ([y_j])$ is an AND or MIX merge point **then**
        Get max time t from $[x_k] \rightarrow ([y_j])$
        Append to $Tr(t)$ the branch $[x_k]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
        if $\rightarrow ([y_j])$ is a MIX or OR merge point **then**
          Append to $Tr(t)$ the branch $[x]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
        end if
      end if
    end if
  end if
end for
$i++$
end while=0

Algorithm 2 $Check(x)$ [1]: Determines if there exists some $y \in V$ where $(x, y) \in E$ and its attribute $L((x, y))$ allows that at least one traversal on the arc. If y exists, the algorithm updates $T((x, y))$ and returns y, otherwise it returns $\emptyset$.

Input: $x \in V$
Output: $y \in V$ if the arc attribute $L((x, y))$ allows that at least one traversal on the arc, $\emptyset$ otherwise
1: $y \leftarrow w$ where $w \in V$, $(x, w) \in E$ and for $1 \leq i \leq |L(x, w)|$ such that either $CTInd_{(x,y)}[i-1] = 2$ and $CTInd_{(x,y)}[i] = 0$, or $CTInd_{(x,y)}[i = 1] = 0$
2: **if** $y \neq \emptyset$ **then**

3: **if** $\exists u \in V$ such that $(u, x) \in E$ **then**
4: $t_i \in T((x, y)) : t_i \leftarrow maxV + 1$ where $maxV = \max_{(u,x)\in E} \{max\{t_k | 1 \leq k \leq L((u, x)),\ t_k \in T((u, x))\}\} + 1$
5: **else**
6: $t_i \in T((x, y)) : t_i \leftarrow 1$
7: **end if**
8: **return** y
9: **else**
10: **return** $\emptyset$
11: **end if**=0

Theorem 5. *Given an RDLT R, a maximal activity S is derivable from R using Algorithm A in [1] if and only if there exists a branch in the traversal tree that corresponds to the reachability sets of S, that is, at time step i, both A and Algorithm 1 traverse an arc and a branch respectively [2].*

Proof. The different structures that compose an RDLT are used to prove this theorem. We first prove that a maximal activity S derived from Algorithm A in [1] form the RDLT is consistent with a branch in the traversal tree. Then we establish that a branch in the generated traversal tree can be successfully derived as a maximal activity S using the Algorithm A in [1].

Structure 1 is two vertices x and y connected through an ϵ-constrained arc from x to y. The corresponding branch of this structure will create a vertex x and y, along with a line connecting x to y with an arrowhead pointing towards y. The branch will be connected as follows: $x_{S[0,...]} \rightarrow y_{S[0,...,\epsilon]}$. If y is reachable from x in R, there would exist an activity profile $S = \{\ldots, S(i) = \{(x, y)\}, \ldots\}$. In the traversal tree, the connection of vertices from $x \rightarrow y$ is at a time step $Tr(i)$, with $y_{S[0,...,\epsilon]}$ having the satisfied condition in ϵ-constraint, traversing from vertex $x_{S[0,...]}$ (shown in Figure 9).

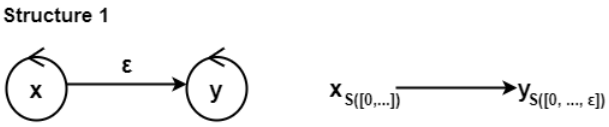

Fig. 9. Structure 1 of an RDLT and its corresponding branch in the traversal tree

Structure 2 is two vertices x and y connected through an Σ-constrained arc from x to y. The corresponding branch of this structure will create a vertex x and y, along with a line connecting x to y with an arrowhead pointing towards y. The branch will be connected as follows: $x_{S[0,...]} \rightarrow y_{S[0,...,a]}$. If y is reachable

from x in R, there would exist an activity profile $S = \{\ldots, S(i) = \{(x, y)\}, \ldots\}$. In the traversal tree, the connection of vertices from $x \to y$ is at a time step $Tr(i)$, with $y_{S[0,\ldots,a]}$ having the satisfied condition in Σ-constraint, traversing from vertex $x_{S[0,\ldots]}$ (shown in Figure 10).

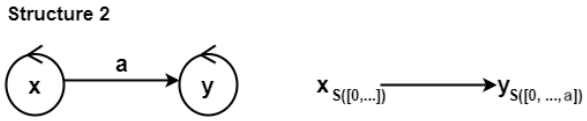

Fig. 10. Structure 2 of an RDLT and its corresponding branch in the traversal tree

Structure 3 is three vertices x, y, and z connected through an ϵ-constrained arc from x to y and x to z. This is called a SPLIT at vertex x. As per the rule for every split in generating parallel activities, we induce a branch per split. The corresponding branches of this structure will create $x \to y$, and $x \to z$. In this scenario, we could have 3 possible cases;

1. Case 1: $S = \{\ldots, S(i) = \{(x, y)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \to y_{S[0,\ldots,\epsilon]}$ at $Tr(i)$.
2. Case 2: $S = \{\ldots, S(i) = \{(x, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \to z_{S[0,\ldots,\epsilon]}$ at $Tr(i)$.
3. Case 3: $S = \{\ldots, S(i) = \{(x, y), (x, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \to y_{S[0,\ldots,\epsilon]}$ and $x_{S[0,\ldots]} \to z_{S[0,\ldots,\epsilon]}$ at $Tr(i)$.

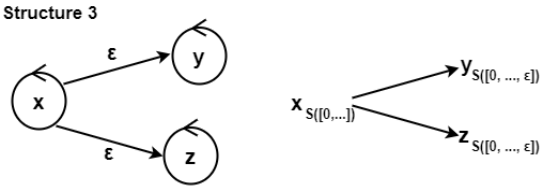

Fig. 11. Structure 3 of an RDLT and its corresponding branch in the traversal tree

Structure 4 is three vertices x, y, and z connected through a Σ-constrained arc from x to y and x to z. This is also a SPLIT at vertex x. As per the rule for every split in generating parallel activities, we induce a branch per split. The corresponding branches of this structure will create $x \to y$, and $x \to z$. In this scenario, we could have 3 possible cases;

1. Case 1: $S = \{\ldots, S(i) = \{(x, y)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \rightarrow y_{S[0,\ldots,a]}$ at $Tr(i)$.
2. Case 2: $S = \{\ldots, S(i) = \{(x, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \rightarrow z_{S[0,\ldots,b]}$ at $Tr(i)$.
3. Case 3: $S = \{\ldots, S(i) = \{(x, y), (x, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \rightarrow y_{S[0,\ldots,a]}$ and $x_{S[0,\ldots]} \rightarrow z_{S[0,\ldots,b]}$ at $Tr(i)$.

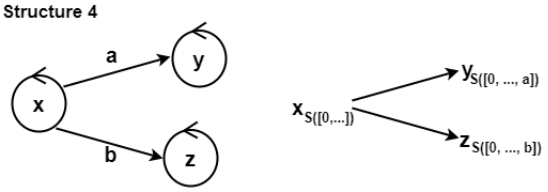

Fig. 12. Structure 4 of an RDLT and its corresponding branch in the traversal tree

Structure 5 is three vertices x, y, and z connected through a Σ-constrained arc from x to y, and an ϵ-constrained arc from x to z. This is also a SPLIT at vertex x. As per the rule for every split in generating parallel activities, we induce a branch per split. The corresponding branches of this structure will create $x \rightarrow y$, and $x \rightarrow z$. In this scenario, we could have 3 possible cases;

1. Case 1: $S = \{\ldots, S(i) = \{(x, y)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \rightarrow y_{S[0,\ldots,a]}$ at $Tr(i)$.
2. Case 2: $S = \{\ldots, S(i) = \{(x, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \rightarrow z_{S[0,\ldots,\epsilon]}$ at $Tr(i)$.
3. Case 3: $S = \{\ldots, S(i) = \{(x, y), (x, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \rightarrow y_{S[0,\ldots,a]}$ and $x_{S[0,\ldots]} \rightarrow z_{S[0,\ldots,\epsilon]}$ at $Tr(i)$.

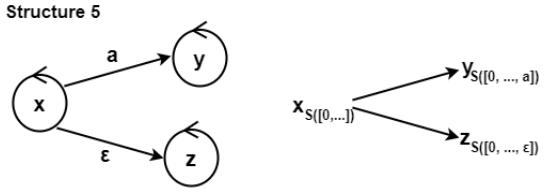

Fig. 13. Structure 5 of an RDLT and its corresponding branch in the traversal tree

Structure 6 is three vertices x, y, and z connected through a ϵ-constrained arc from x to z and y to z. This is called an OR-join at vertex z. As per the

rule for an OR-join in generating parallel activities, the flow of the process will not wait for any of the merging arc to reach z. The corresponding tree of this structure will create $x \to z$, and $y \to z$. In this scenario, we could have 2 possible cases;

1. Case 1: $S = \{\ldots, S(i) = \{(x, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,\epsilon]}$ at $Tr(i)$.
2. Case 2: $S = \{\ldots, S(i) = \{(y, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $y_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,\epsilon]}$ at $Tr(i)$.
3. Case 3: $S = \{\ldots, S(i) = \{(x, z), (y, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,\epsilon]}$ at $Tr(i)$, and $y_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,\epsilon]}$ at $Tr(i)$.

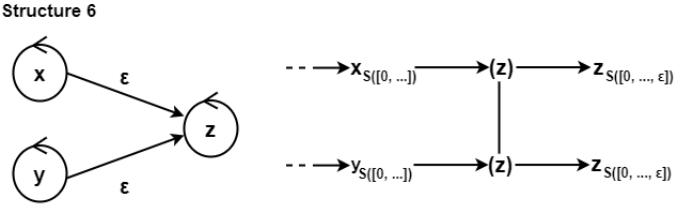

Fig. 14. Structure 6 of an RDLT and its corresponding branch in the traversal tree

Structure 7 is three vertices x, y, and z connected through a Σ-constrained arc from x to z and y to z. This is called an AND-join at vertex z. As per the rule for an AND-join in generating parallel activities, the joining of branches at z will merge to a branch. The corresponding tree of this structure will create $x \to (z) \to z$, and $y \to (z) \to z$. In this scenario, we could have 3 possible cases;

1. Case 1: $S = \{\ldots, S(i) = \{(w, x)\}, \ldots, S(i + n) = \{(y, z)\}\}$, where x has already been reached by some vertex w and $n \in \mathbb{N}$. In the traversal tree, there would exist a branch $w_{S[0,\ldots]} \to x_{S[0,\ldots]}$ at $Tr(i)$, $y_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,(a,b)]}$ at $Tr(i + n)$, where $i \in \mathbb{N}$.
2. Case 2: $S = \{\ldots, S(i) = \{(w, y)\}, \ldots, S(i + n) = \{(x, z)\}\}$, where y has already been reached by some vertex w and $n \in \mathbb{N}$. In the traversal tree, there would exist a branch $w_{S[0,\ldots]} \to y_{S[0,\ldots]}$ at $Tr(i)$, $x_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,(a,b)]}$ at $Tr(i + n)$, where $i \in \mathbb{N}$.
3. Case 3: $S = \{\ldots, S(i) = \{(w, x), (y, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \to (z) \to$ together with $y_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,(a,b)]}$ at $Tr(i + n)$, where $i \in \mathbb{N}$.

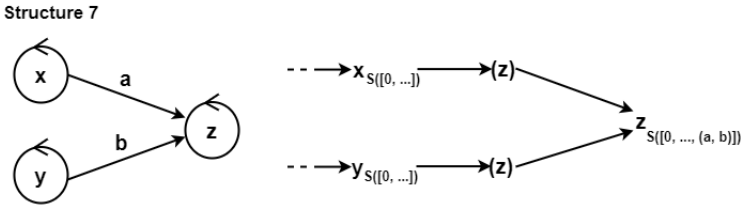

Fig. 15. Structure 7 of an RDLT and its corresponding branch in the traversal tree

Structure 8 is four vertices w, x, y, and z with a Σ-constrained arc from w to z, a Σ-constrained arc from x to z and a Σ-constrained arc from y to z where $C'((w,z)) = C'((x,z)) \neq C'((y,z))$. This is a case of an OR-join at $(w,z),(x,z)$, and finally merges at an AND-join at (y,z). As per the rule for OR- and AND-join in generating parallel activities, the corresponding tree of this structure will create $w \to (z) \to z$, $x \to (z) \to z$, and finall a $y \to (z) \to z$. In this scenario, we could have 3 possible cases;

1. Case 1: $S = \{\ldots, S(i) = \{(u,w)\}, \ldots, S(i+n) = \{\{(y,z)\}\}$, where w has already been reached by some vertex u and $n \in \mathbb{N}$. In the traversal tree, there would exist a branch $u_{S[0,\ldots]} \to w_{S[0,\ldots]}$ at $Tr(i)$, $y_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,(a,b)]}$ at $Tr(i+n)$, where $i \in \mathbb{N}$.
2. Case 2: $S = \{\ldots, S(i) = \{(u,x)\}, \ldots, S(i+n) = \{\{(y,z)\}\}$, where w has already been reached by some vertex u and $n \in \mathbb{N}$. In the traversal tree, there would exist a branch $u_{S[0,\ldots]} \to x_{S[0,\ldots]}$ at $Tr(i)$, $y_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,(a,b)]}$ at $Tr(i+n)$, where $i \in \mathbb{N}$.
3. Case 3: $S = \{\ldots, S(i) = \{(w,z),(x,z),(y,z)\}, \ldots\}$. In the traversal tree, there would exist a branch $w_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,(a,b)]}$ at $Tr(i)$ and a branch $x_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,(a,b)]}$ at $Tr(i)$, where $i \in \mathbb{N}$.

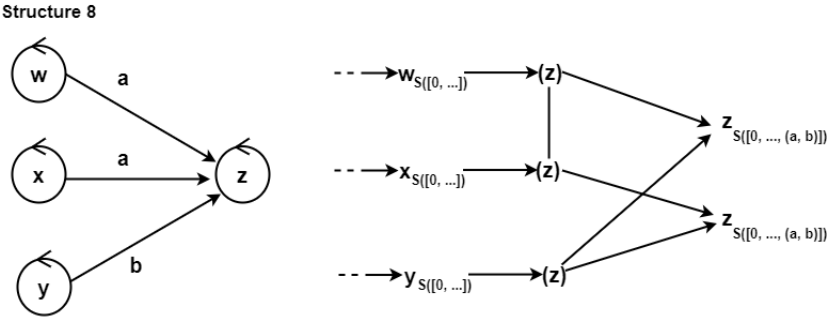

Fig. 16. Structure 8 of an RDLT and its corresponding branch in the traversal tree

Structure 9 is three vertices x, y, and z connected through an ϵ-constrained arc from x to z, and a Σ-constrained arc from y to z. This is called a MIX-join at vertex z. As per the rule for a MIX-join in generating parallel activities, the corresponding tree of this structure will create $x \to (z) \to z$, and $y \to (z) \to z$. In this scenario, we could have 3 possible cases:

1. Case 1: $S = \{\ldots, S(i) = \{(y, z)\}\}$. In the traversal tree, there would exist a branch $y_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,b]}$ at $Tr(i)$, where $i \in \mathbb{N}$.
2. Case 2: $S = \{\ldots, S(i) = \{(w, y)\}, \ldots, S(i+n) = \{(x, z)\}\}$ where y has already been reached by some vertex w and $n \in \mathbb{N}$. In the traversal tree, there would exist a branch $u_{S[0,\ldots]} \to y_{S[0,\ldots]}$ at $Tr(i)$, and $x_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,(\epsilon,b)]}$ at $Tr(i+n)$, where $i \in \mathbb{N}$.
3. Case 3: $S = \{\ldots, S(i) = \{(x, z), (y, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $x_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,(\epsilon,b)]}$ together with $y_{S[0,\ldots]} \to (z) \to z_{S[0,\ldots,b]}$ at $Tr(i)$, where $i \in \mathbb{N}$.

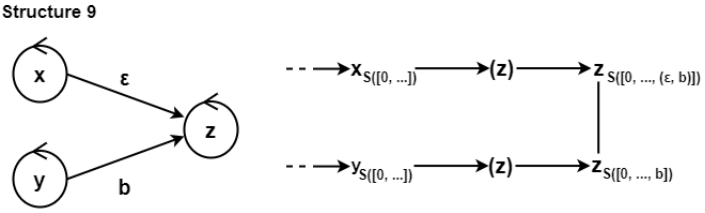

Fig. 17. Structure 9 of an RDLT and its corresponding branch in the traversal tree

If a vertex y is reachable from a vertex x in the RDLT, then there exists an activity profile such that $S = \{\ldots, S(i) = \{(x, x_1), \ldots\}, \ldots, S(i+n) = (x_n, y), \ldots\}$ where $n \in \mathbb{N}$. The set of reachability configurations $S(i)$ to $S(i+n)$ can be divided into different parts each corresponding to one of the structures shown above. Thus, the branch at a certain $Tr(i)$ is obtained by following the rules in producing separate and merged activity sets of each structure.

Theorem 6. *The space and time complexity of Algorithm 1 is* $O(2^n)$*, where* $n = |V| + |E|$*,* $|V|$ *represents the total count of vertices,* $|E|$ *as the total count of edges [2].*

Proof. The space complexity of Algorithm 1 is primarily set on the storage of the traversal tree. Each vertex in the traversal tree is stored along with its outgoing branches. The space complexity of the algorithm could exponentially grow as it reaches every vertex and edge. Thus, the overall space complexity of the algorithm can be expressed as $O(2^n)$ in generating the branches of maximal activities in R, where $n = |V| + |E|$, $|V|$ represents the total count of vertices, $|E|$ as the total count of edges.

For the time complexity of Algorithm 1, the while loop iterates until there are no more vertex to explore, which is determined by the condition $X = \emptyset$. Every vertex in the input RDLT might be needed to be explored. Therefore, the time complexity for traversing the vertices can be expressed as $O(|V|)$, such that $|V|$ is the total count of vertices. For each vertex $x \in X$, the algorithm processes its outgoing edges. In the worst-case scenario, each vertex has multiple outgoing edges. Therefore, the complexity for the time of processing outgoing edges can be expressed as $O(|E|)$, where $|E|$ is the total count of edges.

Resolving merge points involves additional operations, such as determining the maximum time and adding branches to the traversal tree. This operation occurs within the loop for processing outgoing edges and can be considered constant time for each iteration. In the worst case, the time complexity of the algorithm will exponentially grow as it reaches every vertex and edge.

Thus, the time complexity of Algorithm 1 can be expressed as $O(2^n)$ in generating the branches of maximal activities in R, where $n = |V| + |E|$, $|V|$ represents the total count of vertices, $|E|$ as the total count of edges.

C Modified GTT Algorithm for ERDLT

Algorithm 3 Generation of Traversal Tree for ERDLT, $Tr(t)$, $t = 1, 2, ..., k$, $k \in \mathbb{N}$. Assumption per branch: Independent treatment of maximal activities when performing checks and traversals.

Input: ERDLT R, with one source vertex s and one sink vertex f
Output: Traversal tree, set of parallel and non-parallel maximal activities

Set $i = 1$
Set $Tr(1) = [s]_{S([0])}$
Set $Ancestors = \emptyset$
while $TRUE$ **do**
  Let X be the set of vertices in Tr with no outgoing branches pointing to ancestral vertices, $x \in X$, $x \neq pending$, i.e. $([x])$
  if $x \in X$, $x = f$ and no other observed continuing branch **then**
    Set $parallelActivities_p$ to be the set of activities that reaches $[f]$ at the same times, $1 \leq p \leq k$, with $eRU(x, y) \in E \geq$ its actual use
    Set $UnparallelActivities_q$ to be the set of activities that reaches $[f]$ at different times, $1 \leq q \leq k$, with $eRU(x, y) \in E \geq$ its actual use
    return $[Tr(t), parallelActivities_p, UnparallelActivities_q, \forall p, q]$
  else if $X = \emptyset$ **then**
    return 0
  end if
  for each $(x, y_j) \in E$ where $x \in X$ **do**
    if x is a guard vertex, i.e. $V_{type}(x) =$ 'g' **then**
      $CheckGuardCondition(x)$
    else if x is an invoker, i.e. $V_{type}(x) =$ 'i' **then**

Get invoking arc (x, y_j) from invoked ERDLT R' where $y_j \in R'$ is its source, $V_{type}(y) =$ 'b' //shifts branch generation on the same traversal tree from R to R'
Set $prevConstraints = S([0, satisfiedConstraintAncestralArc])$
Set $updatedConstraints = S([0, satisfiedConstraintAncestralArc, currentSatisfiedArcConstraint])$
Append $Tr(i)$ a branch $[x]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
end if
if $Check(x, y_j) = TRUE$ and y_j is in $Ancestors$, such that $eRU(x, y_j) \in E \geq$ its actual use **then**
if (x, y_j) is a returning arc **then**
Assign invoker from R which called the current invoked ERDLT R' to y_j //shifts branch generation on the same traversal tree from R' back to R after this branch
end if
if (x, y_j) is unconstrained **then**
Set $prevConstraints = S([0, satisfiedConstraintAncestralArc])$
Set $updatedConstraints = S([0, satisfiedConstraintAncestralArc, currentSatisfiedArcConstraint])$
Append $Tr(i)$ a branch $[x]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
else
Append $Tr(i)$ a branch $[x] \rightarrow ([y_j])$
Checks for possibility to resolve $\rightarrow ([y_j])$
if $\rightarrow ([y_j])$ is an AND or MIX merge point **then**
Get max time t from $[x_k] \rightarrow ([y_j])$
Append to $Tr(t)$ the branch $[x_k]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
if $\rightarrow ([y_j])$ is a MIX or OR merge point **then**
Append to $Tr(t)$ the branch $[x]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
end if
end if
end if
end if
end for
Append to $Ancestors$ the vertex x
$i++$
end while
while $TRUE$ **do**
Let X be the set of vertices in Tr with no outgoing branches and $x \in X$, x is not a pending vertex, i.e. $([x])$
if $x \in X$, $x = f$ and no other observed continuing branch **then**
Set $parallelActivities_p$ to be the set of activities that reaches $[f]$ at the same times, $1 \leq p \leq k$

Set $UnparallelActivities_q$ to be the set of activities that reaches $[f]$ at different times, $1 \leq q \leq k$
return $[Tr(t), parallelActivities_p, UnparallelActivities_q, \forall p, q]$
else if $X = \emptyset$ **then**
return 0
end if
for each $(x, y_j) \in E$ where $x \in X$ **do**
if x is a guard vertex, i.e. $V_{type}(x) =$ 'g' **then**
$CheckGuardCondition(x)$
else if x is an invoker, i.e. $V_{type}(x) =$ 'i' **then**
Get invoking arc (x, y_j) from invoked ERDLT R' where $y_j \in R'$ is its source, $V_{type}(y) =$ 'b' //shifts branch generation on the same traversal tree from R to R'
Set $prevConstraints = S([0, satisfiedConstraintAncestralArc])$
Set $updatedConstraints = S([0, satisfiedConstraintAncestralArc, currentSatisfiedArcConstraint])$
Append $Tr(i)$ a branch $[x]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
end if
if $Check(x, y_j) = TRUE$ **then**
if (x, y_j) is a returning arc **then**
Assign invoker from R which called the current invoked ERDLT R' to y_j //shifts branch generation on the same traversal tree from R' back to R after this branch
end if
if (x, y_j) is unconstrained **then**
Set $prevConstraints = S([0, satisfiedConstraintAncestralArc])$
Set $updatedConstraints = S([0, satisfiedConstraintAncestralArc, currentSatisfiedArcConstraint])$
Append $Tr(i)$ a branch $[x]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
else
Append $Tr(i)$ a branch $[x] \rightarrow ([y_j])$
Checks for possibility to resolve $\rightarrow ([y_j])$
if $\rightarrow ([y_j])$ is an AND or MIX merge point **then**
Get max time t from $[x_k] \rightarrow ([y_j])$
Append to $Tr(t)$ the branch $[x_k]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
if $\rightarrow ([y_j])$ is a MIX or OR merge point **then**
Append to $Tr(t)$ the branch $[x]_{prevConstraints} \rightarrow [y_j]_{updatedConstraints}$
end if
end if
end if
end if
end for
$i++$

86: **end while**=0

Algorithm 4 *CheckGuardCondition*(x) [3]: Determining whether the condition of a guard x was satisfied or not. If it is satisfied, the algorithm fills -1 to all $T((x,y))$ according to $L((x,y))$ of all such arc (x,y) where $C((x,y))$ is negative, otherwise, -1 is filled to all such arc (x,y) where $C((x,y))$ is positive.

Input: Guard vertex x

Output: $T((x,y)) = [t_1, ..., t_n]$ where $n = L((x,y))$ and every $t_i = -1$ for all negative C-valued outgoing arcs of x if the condition of x is satisfied. Otherwise, $t_i = -1$ for all $0 < i \leq n$ is assigned to all positive C-valued outgoing arcs of x

1: **if** condition of guard x is satisfied **then**
2: $Q \subseteq V : Q \leftarrow \{\forall(x,r) \in E,\ r \in V$ where $C((x,r))$ is negative$\}$ //selects all negative $C-$valued arcs and stores them into set Q
3: **else**
4: $Q \subseteq V : Q \leftarrow \{\forall(x,r) \in E,\ r \in V$ where $C((x,r))$ is positive$\}$ //selects all positive $C-$valued arcs and stores them into set Q
5: **end if**
6: **while** $\exists(x,r) \in Q$ where $|\{t_k \in T((x,r))|t_k \geq 0\}| \geq 0$ **do**
7: $\forall t_k \in T((x,r)) : t_k \leftarrow -1$ //set all $T-$values of each arc in Q according to their respective $L-$values
8: **end while**=0

D Proofs of Theorems in Sections 4.4 and 5

Theorem 1. *Given an ERDLT R and an RDLT R' where R and R' are isomorphic to each other and have equal attribute values, Algorithm 3: MGTT generates its traversal tree from R that is equivalent to the traversal tree generated by Algorithm 1: GTT from using R'.*

Proof. The justification given to the proposed modification to GTT resulted in the addition of code blocks to consider only the presence of invokers (with its corresponding invoking and returning arcs) and guard vertices. These code blocks are in `if` statements that are only executed if the traversed vertex at some time step is an invoker or a guard. Therefore, if there are no such vertices in an ERDLT model R, together with the other unique vertex and arc types of ERDLTs, this implies that the original GTT can be used to generate a traversal tree from R. Specifically, there exists some RDLT R' that is isomorphic to R (see Remark 2 below) with the context given above, which can be an input to the original GTT to generate a traversal tree. Furthermore, the difference between RDLT and ERDLT is their sets of vertices and arcs, and there is no claim in [3,4] that the absence of unique vertex and arc types of ERDLT in R will make R not an ERDLT model. Therefore, we can also call the RDLT model R' in Figure 2 an ERDLT model, or there also exists an ERDLT R that is isomorphic to R'. Consequently, we can execute both the original and modified versions of GTT on

R' and R, respectively. Since the new code blocks in the modified GTT will not execute because there are no invokers or guards in R, both versions will generate trees that are exactly the same. Moreover, if R and R' do not contain the looping arc, i.e. $(z2, y2) \notin E$ and $(z2', y2') \notin E'$, the trees that will be generated by both the original and modified GTT will be equal to the final tree in Figure 8.

□

Remark 2. The isomorphism mentioned in Theorem 1 means that the vertices x and y in an ERDLT R are connected if and only if their corresponding vertices x' and y' in the RDLT R' are connected.

Theorem 2. *Given an ERDLT R, a maximal activity S can be derived from R using the activity extraction algorithm $\mathcal{A}'$ in [4] if and only if there exists a branch in the set of traversal trees that corresponds to the reachability sets (or sets of arcs) of S, that is, at time step i, both the activity extraction algorithm and Algorithm 3: MGTT traverse an arc and a branch, respectively.*

Proof. This proof is patterned on the proof made in [2] for the similar theorem on Algorithm $\mathcal{A}$ in [1] and the original algorithm for generating a traversal tree for RDLT models in [2]. This proof is given in Appendix B. We first prove that a maximal activity S derived from the activity extraction algorithm for an ERDLT model is consistent with a branch in the traversal tree. We then establish that a branch in the generated traversal tree can be derived as a maximal activity S using the activity extraction algorithm for ERDLT in [4].

The basis for whether an arc is traversable is fully reused in ERDLT [3,4]. It is also the basis for the three join structures in RDLT models [1]. Hence, such structures (including splits) can be fully constructed in ERDLT models. This is also reflected in the activity extraction algorithm for ERDLT since it is a modification of algorithm $\mathcal{A}$ in [1], where their difference is only a few additional blocks of codes dealing with guard and invoker vertices. In line with this, the proof presented in the Appendix B would also fully apply here.

In addition to the nine structures in the proof of Theorem 5 in [2] (see Appendix B), we present additional structures for the split types of guard vertices in a traversal tree Tr and for invokers along with the corresponding invoking and returning arcs. For the former, since guard vertices would still block some arcs that correspond to meeting their condition or not, maximal activities that use these blocked arcs can only be represented in a separate traversal tree Tr' where these arcs would not be blocked upon traversing the guard vertex.

Structure 10 is three vertices x, y, and z connected through a true arc (x, y) and a false arc (x, z). This is called a (strictly) XOR-split at vertex x. As per the rule for such split structure, the guard proceeds to block a signed arc depending on whether its condition is met or not. This will produce two scenarios on different traversal trees:

– If the guard condition is met, the corresponding branch of this structure will create a vertex x and y, along with a line connecting x to y with an arrowhead pointing towards y. The branch will be connected as follows:

$x_{S[0,...]} \rightarrow y_{S[0,...,+\epsilon]}$. If y is reachable from x in R, there would exist an activity profile $S = \{\ldots, S(i) = \{(x, y)\}, \ldots\}$. In the traversal tree, the connection of vertices from $x \rightarrow y$ is at a time step $Tr(i)$, with $y_{S[0,...,+\epsilon]}$ having the satisfied condition in C of (x, y), traversing from vertex $x_{S[0,...]}$.

– If the guard condition is not satisfied, the corresponding branch of this structure will create a vertex x and z, along with a line connecting x to z with an arrowhead pointing towards z. The branch will be connected as follows: $x_{S[0,...]} \rightarrow z_{S[0,...,-\epsilon]}$. If z is reachable from x in R, there would exist an activity profile $S = \{\ldots, S(i) = \{(x, z)\}, \ldots\}$. In the traversal tree, the connection of vertices from $x \rightarrow z$ is at a time step $Tr(i)$, with $z_{S[0,...,-\epsilon]}$ having the satisfied condition in C of (x, z), traversing from vertex $x_{S[0,...]}$.

Figure 18 shows this split structure and its corresponding two possible branches in different traversal trees Tr and Tr'.

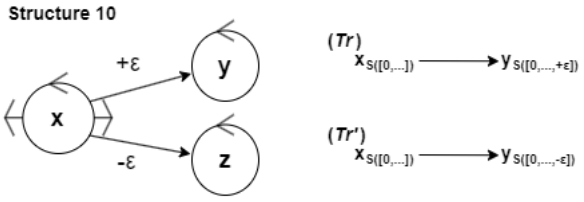

Fig. 18. Structure of a (strictly) XOR-split in ERDLT and its corresponding branch in the traversal tree

Structure 11 is four vertices w, x, y, and z connected through a true arc (w, x) and false arcs (w, y) and (w, z). This is called a partial XOR-split at vertex w. As per the rule for such split structure, the guard proceeds to block signed arcs depending on whether its condition is met or not. This will produce two scenarios on different traversal trees:

– If the guard condition is met, the corresponding branch of this structure will create a vertex w and x, along with a line connecting w to x with an arrowhead pointing towards x. The branch will be connected as follows: $w_{S[0,...]} \rightarrow x_{S[0,...,+\epsilon]}$. If x is reachable from w in R, there would exist an activity profile $S = \{\ldots, S(i) = \{(w, x)\}, \ldots\}$. In the traversal tree, the connection of vertices from $w \rightarrow x$ is at a time step $Tr(i)$, with $x_{S[0,...,+\epsilon]}$ having the satisfied condition in C of (w, x), traversing from vertex $w_{S[0,...]}$.
– If the guard condition is not satisfied, there are two adjacent vertices allowed for traversal. This is still a SPLIT after the guard blocked the non-satisfying arcs. As per the rule for every OR/AND-split in generating parallel activities, we induce a branch per split. The corresponding branches of this structure will create $w \rightarrow y$ and $w \rightarrow z$. In this scenario, we could have three possible cases:

1. Case 1. $S = \{\ldots, S(i) = \{(w, y)\}, \ldots\}$. In the traversal tree, there would exist a branch $w_{S[0,\ldots]} \rightarrow y_{S[0,\ldots,-\epsilon]}$ at $Tr(i)$.
2. Case 2. $S = \{\ldots, S(i) = \{(w, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $w_{S[0,\ldots]} \rightarrow z_{S[0,\ldots,-\epsilon]}$ at $Tr(i)$.
3. Case 3. $S = \{\ldots, S(i) = \{(w, y), (w, z)\}, \ldots\}$. In the traversal tree, there would exist a branch $w_{S[0,\ldots]} \rightarrow y_{S[0,\ldots,-\epsilon]}$ and $w_{S[0,\ldots]} \rightarrow z_{S[0,\ldots,-\epsilon]}$ at $Tr(i)$.

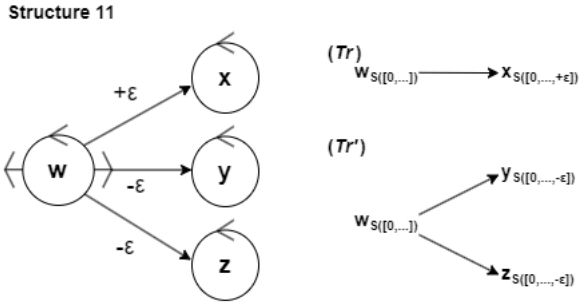

Fig. 19. Structure of a partial XOR-split with one true arc and multiple false arcs in ERDLT and its corresponding branch in the traversal tree

Structure 12 is four vertices w, x, y, and z connected through true arcs (w, x) and (w, y) and a false arc (w, z). This is called a partial XOR-split at vertex w. As per the rule for such split structure, the guard proceeds to block signed arcs depending on whether its condition is met or not. This will produce two scenarios on different traversal trees:

– If the guard condition is met, there are two adjacent vertices allowed for traversal. This is still a SPLIT after the guard blocked the non-satisfying arcs. As per the rule for every OR/AND-split in generating parallel activities, we induce a branch per split. The corresponding branches of this structure will create $w \rightarrow x$ and $w \rightarrow y$. In this scenario, we could have three possible cases:
  1. Case 1. $S = \{\ldots, S(i) = \{(w, x)\}, \ldots\}$. In the traversal tree, there would exist a branch $w_{S[0,\ldots]} \rightarrow x_{S[0,\ldots,+\epsilon]}$ at $Tr(i)$.
  2. Case 2. $S = \{\ldots, S(i) = \{(w, y)\}, \ldots\}$. In the traversal tree, there would exist a branch $w_{S[0,\ldots]} \rightarrow y_{S[0,\ldots,+\epsilon]}$ at $Tr(i)$.
  3. Case 3. $S = \{\ldots, S(i) = \{(w, x), (w, y)\}, \ldots\}$. In the traversal tree, there would exist a branch $w_{S[0,\ldots]} \rightarrow x_{S[0,\ldots,+\epsilon]}$ and $w_{S[0,\ldots]} \rightarrow y_{S[0,\ldots,+\epsilon]}$ at $Tr(i)$.
– If the guard condition is not satisfied, the corresponding branch of this structure will create a vertex w and z, along with a line connecting w to z with an arrowhead pointing towards z. The branch will be connected as follows:

$w_{S[0,...]} \rightarrow z_{S[0,...,-\epsilon]}$. If z is reachable from w in R, there would exist an activity profile $S = \{\ldots, S(i) = \{(w, z)\}, \ldots\}$. In the traversal tree, the connection of vertices from $w \rightarrow z$ is at a time step $Tr(i)$, with $z_{S[0,...,-\epsilon]}$ having the satisfied condition in C of (w, z), traversing from vertex $w_{S[0,...]}$.

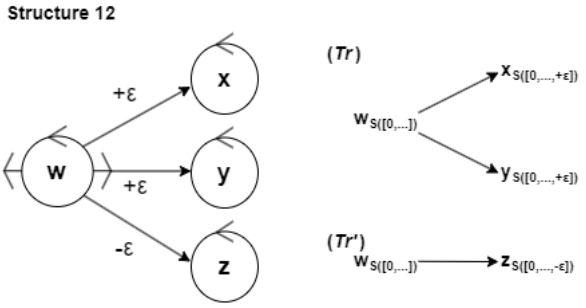

Fig. 20. Structure of a partial XOR-split with one false arc and multiple true arcs in ERDLT and its corresponding branch in the traversal tree

Structure 13 is five vertices z, a, b, c, and d connected through true arcs (z, a) and (z, b) and false arcs (z, c) and (z, d). This is called a pseudo XOR-split at vertex z. As per the rule for such split structure, the guard proceeds to block signed arcs depending on whether its condition is met or not. This will produce two scenarios on different traversal trees:

- If the guard condition is met, there are two adjacent vertices allowed for traversal. This is still a SPLIT after the guard blocked the non-satisfying arcs. As per the rule for every OR/AND-split in generating parallel activities, we induce a branch per split. The corresponding branches of this structure will create $z \rightarrow a$ and $z \rightarrow b$. In this scenario, we could have three possible cases:
  1. Case 1. $S = \{\ldots, S(i) = \{(z, a)\}, \ldots\}$. In the traversal tree, there would exist a branch $z_{S[0,...]} \rightarrow a_{S[0,...,+\epsilon]}$ at $Tr(i)$.
  2. Case 2. $S = \{\ldots, S(i) = \{(z, b)\}, \ldots\}$. In the traversal tree, there would exist a branch $z_{S[0,...]} \rightarrow b_{S[0,...,+\epsilon]}$ at $Tr(i)$.
  3. Case 3. $S = \{\ldots, S(i) = \{(z, a), (z, b)\}, \ldots\}$. In the traversal tree, there would exist a branch $z_{S[0,...]} \rightarrow a_{S[0,...,+\epsilon]}$ and $z_{S[0,...]} \rightarrow b_{S[0,...,+\epsilon]}$ at $Tr(i)$.
- If the guard condition is not satisfied, there are two adjacent vertices allowed for traversal. This is still a SPLIT after the guard blocked the non-satisfying arcs. As per the rule for every OR/AND-split in generating parallel activities, we induce a branch per split. The corresponding branches of this structure will create $z \rightarrow c$ and $z \rightarrow d$. In this scenario, we could have three possible cases:

1. Case 1. $S = \{\ldots, S(i) = \{(z, c)\}, \ldots\}$. In the traversal tree, there would exist a branch $z_{S[0,\ldots]} \rightarrow c_{S[0,\ldots,-\epsilon]}$ at $Tr(i)$.
2. Case 2. $S = \{\ldots, S(i) = \{(z, d)\}, \ldots\}$. In the traversal tree, there would exist a branch $z_{S[0,\ldots]} \rightarrow d_{S[0,\ldots,-\epsilon]}$ at $Tr(i)$.
3. Case 3. $S = \{\ldots, S(i) = \{(z, c), (z, d)\}, \ldots\}$. In the traversal tree, there would exist a branch $z_{S[0,\ldots]} \rightarrow c_{S[0,\ldots,-\epsilon]}$ and $z_{S[0,\ldots]} \rightarrow d_{S[0,\ldots,-\epsilon]}$ at $Tr(i)$.

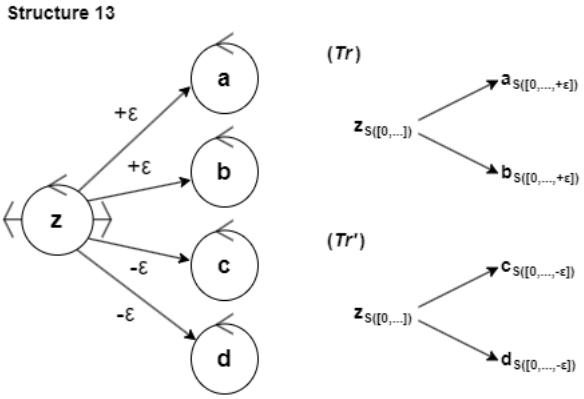

Fig. 21. Structure of a pseudo XOR-split in ERDLT and its corresponding branch in the traversal tree

Structure 14 is three vertices x, y, and a where y is an invoker in a caller ERDLT R together with a vertex x connected with an arc from x to y, while a is the source boundary object of the invoked ERDLT R' pointed at by an invoking arc. The corresponding branch of this structure will create vertices x, y, and a, along with lines connecting x to y and y to a with arrowheads pointing to y and a, respectively. The branch will be connected as follows: $x_{s[0,\ldots]} \rightarrow y_{s[0,\ldots,\epsilon]} \rightarrow a_{s[0,\ldots,\epsilon,\epsilon]}$. If y is reachable from x in R where a is automatically reachable from y with the invoking arc, there would exist activity profiles $S = \{..., S(i) = \{(x, y)\}, S(i + 1) = \{S'\}, ...\}$ where $S' = \{S'(1) = (a, v), ...\}$ and $v \in V'$ in R'. The reachability configurations in S only contain arcs in R, while that of S' only contains arcs in R', but the two combined can be simplified as one activity profile $S_{simp} = \{..., S(i) = \{(x, y)\}, S(i + 1) = \{(y, a)\}, S(i + 2) = \{(a, v)\}...\}$. The arc (y, a) therefore represents the invoking arc in R'. In the traversal tree, the connection of vertices from $x \rightarrow y$ is at time step $Tr(i)$ with $y_{s[0,\ldots,\epsilon]}$ traversed from $x_{s[0,\ldots]}$, while the invoking arc represented by $y \rightarrow a$ is at time step $Tr(i+1)$ with $a_{s[0,\ldots,\epsilon,\epsilon]}$ traversed from $y_{s[0,\ldots,\epsilon]}$, shown in Figure 22.

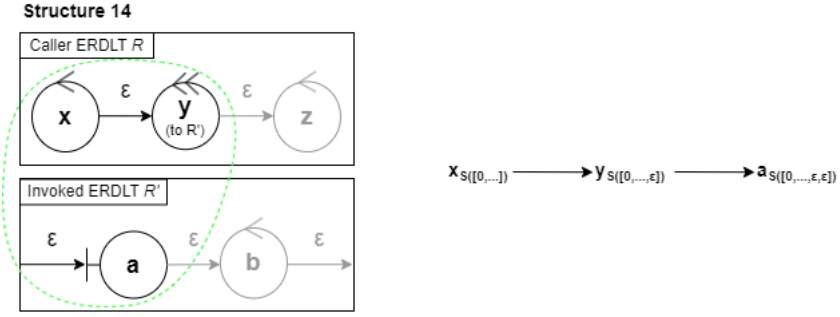

Fig. 22. Structure of an invoker in R with its invoking arc in R' and their corresponding branches in the traversal tree

Structure 15 is three vertices b, y, and z where b is a vertex in an invoked ERDLT R' with an outgoing arc that is a returning arc, while y is an invoker in the caller ERDLT R together with a vertex z connected with an arc from y to z. The corresponding branch of this structure will create vertices b, y, and z, along with lines connecting b to y and y to z with arrowheads pointing to y and z, respectively. The branch will be connected as follows: $b_{s[0,...]} \rightarrow y_{s[0,...,\epsilon]} \rightarrow z_{s[0,...,\epsilon,\epsilon]}$. If z is reachable from y in R where y is automatically reachable from b with the returning arc, there would exist activity profiles $S' = \{..., S'(i) = (b, R^{-1})\}$ and $S = \{..., S(i+1) = \{(y, z)\}, ...\}$ where (b, R^{-1}) represents the returning arc. The reachability configurations in S only contain arcs in R, while that of S' only contains arcs in R', but the two combined can be simplified as one activity profile $S_{simp} = \{..., S(i) = \{(b, y)\}, S(i+1) = \{(y, z)\}, ...\}$. The symbol R^{-1} from the previous representation of the returning arc in S' is therefore replaced by the invoker, hence the arc (b, y) in S_{simp}. In the traversal tree, the returning arc represented by $b \rightarrow y$ is at time step $Tr(i)$ with $y_{s[0,...,\epsilon]}$ traversed from $b_{s[0,...]}$, while the connection of vertices from $y \rightarrow z$ is at time step $Tr(i+1)$ with $z_{s[0,...,\epsilon,\epsilon]}$ traversed from $y_{s[0,...,\epsilon]}$, shown in Figure 23.

□

Theorem 3. *The space and time complexity of Algorithm 3: MGTT is* $O(2^n)$*, where* $n = |V| + |E|$*,* V *and* E *are the sets of vertices and edges in an ERDLT* $R = \{V, E, T, M\}$*.*

Proof. The only addition to the modified version of GTT is dealing with invokers and guard vertices. The former case treats invoking and returning arcs as if the vertices in the invoked ERDLT are directly connected to its caller ERDLT through the invoker. On the other hand, in the latter case, either true or false arcs can only be given a branch in the traversal tree and never both for the same tree. This implies that the guard vertices in an ERDLT model R can reduce the total of possible arcs to traverse. However, it is still possible for an R to contain no invokers or guard vertices. Hence, we follow the proof for Theorem 6 on the

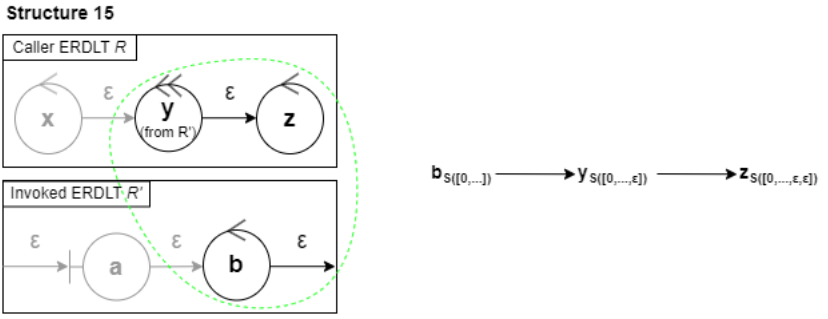

Fig. 23. Structure of an invoker in R with its returning arc in R' and their corresponding branches in the traversal tree

space and time complexity of the original GTT algorithm (see Appendix B) to prove this theorem.

□

Theorem 4. *For the set of traversal trees of an ERDLT model R generated by Algorithm 3: MGTT, the parallel activities of R can be found only among the maximal activities in the same traversal tree.*

Proof. The second requirement for maximal activities to be parallel in Definition 1 states that they should not compete with each other such that their shared resources should be able to accommodate all of them when executed concurrently. Meanwhile, guard vertices impose mutually exclusive paths, as described in Section 4.2. The blocking mechanism of guards on non-satisfying arcs is implemented in the modified generation of traversal tree algorithm (see Algorithm 3 in Appendix C), which causes an ERDLT model to have multiple traversal trees depending on the number of guards present in its set of vertices. There will be at least one maximal activity per tree, similar to the trees in Figure 7. These maximal activities of the ERDLT model R in Figure 5 are spread across these trees so that they are grouped (or isolated) based on the path allowed or blocked by the guards. This implies that a pair of maximal activities of R that do not belong to the same traversal tree each has traversed paths that are mutually exclusive imposed by some guards. Therefore, such pairs should not execute simultaneously in R to preserve the mutual exclusiveness of the paths of the guard vertices. Hence, with the same reasoning, we say that the maximal activities of R that can execute in parallel can only be found among those in the same traversal tree.

□