MutRoSe Semantics Formalization

May 31, 2022

Here we will present a formalization for MutRoSe's mission decomposition process. We will provide construction rules of the multiple structures using operational semantics. Furthermore, we will cover some properties that must hold for the process to be considered correct and will provide arguments using the defined rules in order to show that these properties hold. In the beginning of the section we will provide the definitions that are used throughout the section and then we will provide rules and explanations by dividing the process in four steps: (i) Runtime Annotation Tree and Task Graph generation, (ii) constraints generation, (iii) task instances generation and (iv) valid mission decompositions generation.

1 Definitions

We have several definitions that are used throughout the operational semantics descriptions of the rules and in this section we will provide a description for each of these definitions.

1.1 General Definitions

- T_R is the set of tasks in the Runtime Annotation Tree
- $succ(n_2, n_1)$ is an expression which evaluates to True if n_2 is a child of n_1 and to False otherwise. Being a child means that exists a path from n_1 to n_2
- $Cn(n)$ is an expression that represents the set of children of a node n. This consists of every node m for which $succ(m, n)$ holds. Additionally, $Cn(n)$ optionally returns the number of children that node n has
- $Flat$ is the flattening process which is performed in non-leaf nodes in the Goal Model (GM)
- $F_A(n, k)$ is the forall operator which creates k uniquely identified instances of node n
- $C \vdash n$ means that a context C is satisfied before a certain node n is executed (i.e., it and all of its children are executed), where $\nvdash$ is used if the context is not satisfied
- $C \Vdash n$ means that a context C is satisfied after a certain node n is executed (i.e., it and all of its children are executed) and not satisfied before its execution
- $Nonleaf(n)$ evaluates to True iff n is a non-leaf node (i.e., has at least one outgoing edge)
- $Leaf(n)$ evaluates to True iff n is a leaf node (i.e., has no outgoing edges)

1.2 Runtime Annotation Operators

- $*_{AND}(n_1, n_2)$ where $* \in \{\#, ;, FALLBACK\}$ means that two nodes n_1 and n_2 are AND decomposed by the operator $*$
- $\#_{OR}(n_1, n_2)$ means that two nodes n_1 and n_2 are OR decomposed, being thus inherently parallel (#)

1.3 Goal Model Related

- Q is the set of query goals
- A is the set of non-universal achieve goals
- UA is the set of achieve goals
- T_{GM} is the set of task nodes

1.4 Task Graph Related

- T_G is the set of tasks in the Task Graph
- $T_{OP} = T_{AOP} \cup T_{OOP}$ is the set of operator nodes in the Task Graph where T_{AOP} is the set of AND-decomposed operator nodes and T_{OOP} is the set of OR-decomposed operator nodes
- OP_G is the set of operator nodes in the Task Graph
- D is the set of task decompositions in the Task Graph
  - $Pre(d)$ is the set of boolean preconditions of a decomposition $d \in D$
  - $Eff(d)$ is the set of boolean effects of a decomposition $d \in D$
  - $Fpre(d)$ is the set of function preconditions of a decomposition $d \in D$
  - $Feff(d)$ is the set of function effects of a precondition $d \in D$
- E is the set of edges of a Task Graph TG
- E_{CD} is the set of context dependency edges of a Task Graph TG
- E_{EC} is the set of execution constraint edges of a Task Graph TG
- E_{AND} is the set of AND edges of a Task Graph TG
- E_{OR} is the set of OR edges of a Task Graph TG
- $\leftrightarrow$ ($E_{=}$) represents the creation of bidirectional execution constraint edges
- $\rightarrow$ (E_C) represents the creation of a context dependency edge
- $DC(t)$ is an expression that returns the set of decompositions of a task given given its task ID t

1.5 Constraints Definitions

- C is the constraint set
- A constraint can be defined as the tuple $CTR = (N1, N2, T)$ where:
  - $N1$ is the first task/decomposition involved in the constraint
  - $N2$ is the second task/decomposition involved in the constraint
  - T is the type of the constraint, which can be Sequential (SEQ), Parallel (PAR), Fallback (FB) or Execution (EC)
- CT is the constraint tree
- $N = C_N \cup T_N$ is the set of nodes of CT where:
  - C_N is the set of constraint nodes of CT. Each constraint node in $c \in C_N$ has a property Op which consists of a specific runtime annotation operator (#, ;, FALLBACK) and a property CT which represents the constraints set of the node
  - T_N is the set of tasks nodes of CT. Each task node $t \in T_N$ has a unique task ID called TID

1.6 Valid Mission Decompositions

- P is a set of valid mission decompositions
- $I(N)$ or $i(n)$ is the initial state of a specific node N (or n) in the Task Graph
- $E(N)$ or $e(n)$ is the end state of a specific node N (or n) in the Task Graph
- $S \vdash Pre$ represents the satisfaction of a precondition Pre by the state S
- $S \nvdash Pre$ represents the unsatisfaction of a precondition Pre by the state S
- Given a node N in the Task Graph we have the following:
  (a) P'(N) is the set of valid mission decompositions after checking the node and all of its children
  (b) P*(N) are the paths generated by the node, given some initial state i(N). This is only used with task nodes
  (c) P(N) is the set of valid mission decompositions before checking the node and all of its children
- $B = (pr, ps)$ is a boolean predicate, where:
  - pr is a ground predicate (i.e., one where variables were replaced)
  - ps is a boolean flag, which when set to true indicates that the predicate is positive and negative otherwise
- $F = (pr, val, op)$ is a function predicate, where:
  - pr is a ground predicate
  - val is a numeric value (float or integer)
  - op is the operation. For preconditions we have $op \in [=, !=, >]$ and for effects we have $op \in [assign\ (=), increase\ (+), decrease\ (-)]$
- $S = (S_B, S_F, I)$ is the set of states, where
  - S_B is the set of possible boolean states
  - S_F is the set of possible function states
  - $I = I_B \cup I_F$ is the set of initial states where $I_B \subseteq S_B$ and $I_F \subseteq I_F$
- Boolean preconditions and effects are expressed by means of boolean predicates. Boolean states are represented in the same way as boolean preconditions and effects. Function states, on the other hand, are represented as a tuple $(pred, val)$ where $pred$ is a ground predicate and val is a numeric value.
- A valid mission decomposition $p \in P$ can be defined as the tuple $p = (T, S)$ where:
  - T are the task decompositions involved in the valid mission decompositions
  - S is the resulting world state. This results from applying the effects of each task decomposition $t \in p.T$ over the initial state $i(d) \subseteq I$

2 Runtime Annotation Tree and Task Graph Generation Rules

In this section we start providing the base rules from where all of the other sections will stand upon, since every process makes use of the Task Graph structure in some way. In order to do that, let us first start with the rules to generate the Runtime Annotation Tree (R) from the Goal Model (GM).

Rule 1 *If n_i is a non-leaf node in the Goal Model (GM) it is transformed by means of the flattening process Flat in order to generate the Runtime Annotation Tree (R) node r_i while it is directly transformed into r_i if it is a leaf node, given that it is not a query goal.*

$$\frac{n_i \in GM \mid n_i \notin Q \quad r_i \in R \quad (C_i \vdash n_i) \rightarrow r_i}{\text{if } Nonleaf(n_i) \text{ then } r_i = Flat(n_i) \text{ else } r_i}$$

Rule 2 *A forall statement on a GM non-leaf node n_i creates k times the number of instances of itself and its children in the Runtime Annotation Tree (R), where each r_{jk} is uniquely identified.*

$$\frac{n_i, n_j \in GM \quad r_i, r_j \in R \quad (C_i \vdash n_i) \rightarrow r_i \quad n_j \in \{Cn(n_i) \cup n_i\} \quad (C_j \vdash n_j) \rightarrow r_j}{F_A((C_j \vdash n_j), k) \rightarrow \cup r_{jk}}$$

With the previous rules in mind, we can then proceed to define rules for Task Graph generation.

Rule 3 *Given a node r_i of the Task Graph TG, if the initial state does not satisfy r_i's context condition C_i and there exists $r_j \in TG$, which is parallel-AND decomposed with r_i and where $j < i$, which has at least one child for which its effect satisfies r_i's context condition C_i. Thus, $\forall t \in T_G \mid succ(t, r_i)$ we create a context dependency edge $\rightarrow (E_C)$ going from every task decomposition node $d \in D$ for which $succ(d, r_j)$ and $C_i \Vdash d$.*

$$\frac{C_i \nvdash r_i \quad \exists r_j \mid j < i \ \wedge \ \#_{AND}(r_i, r_j) \ \wedge \ \exists c \mid succ(c, r_j) \ \wedge \ C_i \Vdash c \quad r_i, r_j \in TG \quad i, j \in \mathbb{N}}{d \ \rightarrow (E_C) \ t : \forall d \in D, t \in T_G \mid succ(d, r_j) \ \wedge \ C_i \Vdash d \ \wedge \ succ(t, r_i)}$$

Rule 4 *If we have a non-group node (i.e., $r \in R$ and $!r.\gamma$) we create bidirectional execution constraint edges $\leftrightarrow (E_=)$ for every task node $t \in T_G$ that is a child of the node $n \in TG$ which is generated from r.*

$$\frac{r \in R \quad !r.\gamma \quad n \in TG \quad (C_i \vdash r) \rightarrow n \quad \tau \stackrel{\text{def}}{=} \{(t_1, t_2) \in T_G \mid t_1, t_2 \in Cn(n) \ \wedge \ *_{AND}(t_1, t_2)}{t_i \ \leftrightarrow (E_=) \ t_j \mid \forall (t_i, t_j) \in \tau}$$

Before proceeding to the definition of the next rule we define two sets that will be used in it. The first set τ_1 consists of every pair of AND-decomposed task nodes t_1 and t_2 which are children of two nodes n_1 and n_2 and can be defined as follows:

$$\tau_1(n_1, n_2) \stackrel{\text{def}}{=} \{(t_1, t_2) \in T_G \mid t_1, t_2 \in Cn(n_1) \cup Cn(n_2) \ \wedge \ *_{AND}(t_1, t_2)\}$$

The second set τ_2 consists of every pair of AND-decomposed task nodes t_1 and t_2 which are children of a node n_1 and where one is a child of a node n_2 and the other isn't. This set is defined as follows:

$$\tau_2(n_1, n_2) \stackrel{\text{def}}{=} \{(t_1, t_2) \in T_G \mid t_1, t_2 \in Cn(n_1) \ \wedge \ t_1 \notin Cn(n_2) \wedge \ *_{AND}(t_1, t_2)\}$$

With these definitions in mind, we also define the following helper function for creation of execution constraint edges:

$$\xi(n_1, n_2, t_1, t_2, \tau) = t_1 \leftrightarrow (E_=) t_2 \mid \forall (t_1, t_2) \in \tau(n_1, n_2)$$

Now, we can proceed to define Rule 5.

Rule 5 *Assume there is a group ($r_i.\gamma$) and non-divisible ($!r_i.\sigma$) node $r_i \in R$ and a node r_j which is a child of r_i. Assume also that we have two nodes $n_i \in TG$ and $n_j \in TG$ which are the nodes of the Task Graph generated from r_i and r_j, respectively. Then, if we have $r_j.\gamma$ then we create bidirectional group and non-divisible execution constraint edges between all of the task nodes which are children of $n_i, n_j \in TG$. Otherwise, we create bidirectional group and non-divisible execution constraint edges between all of the task nodes which are children of n_i and where at least one node is not a child of n_j.*

$$\frac{r_i, r_j \in R \mid r_i.\gamma, !r_i.\sigma \quad succ(r_j, r_i) \quad n_i, n_j \in TG \quad (C_k \vdash r_k) \rightarrow n_k, k = \{i, j\}}{\text{if } r_j.\gamma \text{ then } \xi(n_i, n_j, t_i, t_j, \tau_1) \text{ else } \xi(n_i, n_j, t_i, t_j, \tau_2)}$$

3 On the Generation of Task Instances

For the generation of task instances we have no additional rules since all the arguments we need to provide can be given using the rules for the Runtime Annotation Tree and Task Graph generation. This is the case since all of the generated task instances and their decompositions already appear in the Task Graph. In order to do this, we will first define the property we want to verify, defined as Property 1.

Property 1 *A task that appears on the Goal Model appears on the generated set of task instances (and vice-versa), given the corresponding task decompositions and Goal Model expansions*

Secondly, we will provide the necesary arguments. From Rule 1 we can verify that every task t_i in the goal model will appear as a node r_i in the Runtime Annotation Tree as it since, even though they are not leaf nodes, the flattening process only changes the structure of operator nodes. In addition, decomposition nodes, which are always leaf nodes, are transformed into Runtime Annotation Tree nodes that are children of r_i. From Rule 2 we can verify that, in the presence of a forall statement in a node up in the hierarchy from r_i we create k unique copies of r_i, which only differ in variable instantiation but refer to the same task. If we do not have a forall statement in any node up in the hierarchy from r_i, this node is simply left as it is. This structure is then followed by the Task Graph, which keeps the Runtime Annotation Tree structure and adds additional edges defined in Rules 3, 4 and 5. With this in mind we can verify that all of the tasks in the Goal Model appear in the task instances given their decompositions, which are defined by the HTN decomposition process.

4 On the Generation of Constraints

Now we will proceed to define all of the rules related to constraints generation. With the rules of the Task Graph generation in hand, we can define Property 2.

Property 2 *A constraint that is present in the Goal Model appears on the set of constraints, given the corresponding task decompositions and Goal Model expansions*

In addition, before defining the rules for constraints generation we first define what is a tree version of the Task Graph, here called TG_{tree}, which can be defined as follows:

$$TG_{tree} = TG' : E' = E \setminus E_{CD} \cup E_{EC}$$

In order to start the constraints generation process we define the generation of a constraint tree, which is a structure used to generate the initial mission constraints. We define the rule for the generation of the constraint tree next.

Rule 6 *Assume there is a node n_1 in the tree version of the Task Graph which is transformed into a node n_2 in the constraint tree. If n_1 is an operator node OP, where $OP \in \{\#, ;, FALLBACK\}$, then n_2 will be a constraint node with operator equal to OP. If n_1 is not an operator node then n_2 will be a task node with identifier TID equal to n_1's TID identifier*

$$\frac{n_1 \in TG_{tree} \quad n_1 \rightarrow n_2 \mid \text{if } n_1 \in OP_G \text{ then } n_2 \in C_N \wedge n_2.OP = n_1 \text{ else } n_2 \in T_N \wedge n_2.TID = n_1.TID}{N' = N \cup n_2}$$

With the generation of the constraint tree we can proceed to define the rules for constraint generation. Constraint nodes are the ones that generate constraints, where its children can be either other constraint nodes or task nodes. The rule for the generation of constraint nodes is Rule 7. This rule is related to the new constraints generated by the constraint node ct and to the fact that the constraint node set of ct (ct.CT) is expanded with all of the constraints of its children that belong to the constraint node set. Also, $C_{gen}()$ is a function that takes a set of nodes and the type of constraints to be generated and generates new constraints accordingly. The details of this function is explained in Appendix A.

Rule 7 *Assume there is a node ct which is a constraint node in the constraint tree and has its constraint set initially empty. In addition, assume a set ϕ which consists in the union of the constraint sets of every node c which is a child of ct and is also a constraint node and a set γ which consists in the union of the constraint sets of every node c which is generated by the special function C_{gen}. In this sense, the constraint set of ct will be the union of ϕ and γ.*

$$\frac{ct \in C_N, ct.CT = \emptyset \quad C_1 = Cn(ct) \cap C_N, C_2 = C_{gen}(Cn(ct), ct.OP) \quad \phi \stackrel{\text{def}}{=} \bigcup^{c \in C_1} c.CT \quad \gamma \stackrel{\text{def}}{=} \bigcup^{c \in C_2} c.CT}{ct.CT = \phi \cup \gamma}$$

At this point, we can already check that Property 1 (P1) is true with respect to sequential and fallback constraints. Rules related to Task Graph generation show us that every task in the GM is in the Task Graph but with a (possibly) different cardinality due to forall statements. The constraint generation rule shows us that nodes that establish constraints in the Task Graph in fact generate these constraints with the correct operators and also that the constraint set always increases, so constraints generated by a node are never erased.

The following step shown in Rule 8 is related to constraint transformation, where constraints between tasks are transformed into constraints between task decompositions. In addition, any context dependencies between parallel constrained tasks are transformed into sequential constraints. The constraint set C is given by union of parallel ($C.PAR$), sequential ($C.SEQ$) and fallback ($C.FB$) constraints sets, i.e., $C = C.PAR \cup C.SEQ \cup C.FB$. Before defining this rule we establish the following helper function:

$$\beta(S, n, c1, c2, t) \stackrel{\text{def}}{=} (S' = (S \setminus n) \cup CTR(c1, c2, t))$$

With this in mind, the next rule can be defined as follows.

Rule 8 *Assume that we have the generated constraint set C and a constraint $c \in C$. Also, assume a decomposition node d_1 that is a child of the first constraint node N1 of c and a decomposition node d_2 that is a child of the second constraint node N2 of c. In this sense, if c is a parallel constraint we create a new sequential constraint between d_1 and d_2 and add it to C if and only if there is a context dependency edge between them in the Task Graph. If c is not parallel, we create a constraint of the given type ct.T between d_1 and d_2 and add it to C.*

$$\frac{c \in C \quad d_1 \in DC(c.N1.TID) \quad d_2 \in DC(c.N2.TID)}{\text{if } c \in C.PAR \text{ then } \beta(C, c, d_1, d_2, SEQ) \text{ iff } \exists\, e = (d_1, d_2) \in E_{CD} \text{ else } \beta(C, c, d_1, d_2, c.T)}$$

The next rule, which is Rule 9, is related to the creation of execution constraints.

Rule 9 *Assume there is an execution constraint edge e between nodes t_1 and t_2 and that there is no execution constraint between these nodes in the constraint set C. Then, an execution constraint between t_1 and t_2 is created, with the respective group and divisible attributes values.*

$$\frac{e = (t_1, t_2) \in E_{EC} \quad c = (t_1, t_2, EC) \notin C}{C' = C \cup CTR(t_1, t_2, EC)}$$

At this point we can check that P1 is true with respect to execution constraints, from Rule 9, and context dependencies (that in the end generate constraints), from Rule 8.

Finally, the next rule is related to the generation of the minimal constraint set where the rules for the MinimalCTR() function to be satisfied are given in Appendix A.

Rule 10 *Assume a constraint c in the constraint set C. Also, assume there are two constraints c_1 and c_2 for which $c \in C$ implies that C is not minimal, where this condition is expressed by the MinimalCTR() function. Thus, we remove c from C in order to achieve the minimal constraint on C.*

$$\frac{c \in C \quad \exists\ c_1, c_2 \mid c \in C \implies !MinimalCTR(C)}{C' = C \backslash c}$$

5 On the Generation of Valid Mission Decompositions

With the rules of the Task Graph generation in hand we can define Property 3, which is defined next.

Property 3 *Valid mission decompositions are correctly generated*

Based on the definitions for valid mission decompositions, provided in Section 1, we can give a more formal definition of property P3. For valid mission decompositions to be correctly generated we have three conditions that need to be satisfied, which are given below.

Condition 1 $\nexists\ p \in P$ *for which two decompositions d_1 and d_2 of the same task t_1 we have $d_1 \in p.T\ \wedge\ d_2 \in p.T$*

Condition 2 $\forall\ t_1, t_2 \in T_G$ *that are AND-decomposed we must have that $\forall d_1 \in Cn(t_1),\ d_2 \in Cn(t_2)\ \exists\ p \in P$ s.t $d_1 \in p.T \wedge d_2 \in p.T$ iff $i(d_1) \vdash Pre(d_1) \cup Fpre(d_1) \wedge i(d_2) \vdash Pre(d_2) \cup Fpre(d_2)$ and their effects do not conflict, i.e., they do not have opposite effects on the same boolean predicate while being parallel decomposed*

Condition 3 $\nexists\ p \in P$ *for which we have two OR-decomposed tasks t_1 and t_2 where $d_1 \in p.T \wedge d_2 \in p.T$ given that $succ(d_1, t_1)\ \wedge\ succ(d_2, t_2)$*

Before defining valid mission decomposition rules, we need to define the rules for two specific operators which are widely used for this formalization: (i) the parallel combination operator $\otimes$, which is used for combining parallel decomposed paths, and (ii) the sequential combination operator $\circ$, which is used for combining sequentially decomposed paths. In order to define the rule for the parallel combination operator we define the following helper function:

$$\eta(p_1, p_2) = \begin{cases} false, \text{ if } \exists\ s' \in p_1.S \text{ s.t. } \exists\ s'' \in p_2.S \mid \begin{cases} s''.pr = s'.pr\ \wedge\ s''.ps \neq s'.ps, \text{if } s' \in S_B \\ s''.pr = s'.pr\ \wedge\ s''.val \neq s'.val, \text{if } s' \in S_F \end{cases} \\ \text{true, otherwise} \end{cases}$$

With these assumptions we define the rule for the parallel operator as follows.

Rule 11 *Assume two valid mission decomposition set P1 and P2. In addition, we create another valid mission decomposition set P3 by the parallel combination of P1 and P2. In this sense, a decomposition path p_3 is created and added in P3 for every combination of $p_1 \in P1$ and $p_2 \in P2$ if only if p_1 and p_2 do not conflict, which is represented by the function η. For every p_3 that is created its decomposition path is given by the union of the decompositions on path p_1 and the ones in p_2 which are not in p_1 while its world state is given by the union of the world states in p_1 and p_2.*

$$\frac{P1, P2 \quad P3 = P1 \otimes P2 \quad \varphi \overset{\text{def}}{=} \{ \bigcup_{\substack{p_2 \in P2 \\ p_1 \in P1}} p_3 = (p_1.T \cup \{p_2.T \setminus p_1.T\}, p_1.S \cup p_2.S) \text{ iff } \eta(p_1, p_2)\}}{P3 = \varphi}$$

By the condition established in the helper function η we can see that the end state for parallel AND-decomposed tasks is consistent, given that we assume that no conflicting effects can exist for the parallel combination to happen. With this in mind, we have the following rule for the sequential combination operator.

Rule 12 *Assume two valid mission decomposition set P1 and P2. In addition, we create another valid mission decomposition set P3 by the sequential combination of P1 and P2. In this sense, a decomposition path p_3 is created and added in P3 for every combination of $p_1 \in P1$ and $p_2 \in P2$. For every p_3 that is created its decomposition path is given by the union of the decompositions on path p_1 and the ones in p_2 which are not in p_1 while its world state is given by the union of the world states in p_1 and the ones in p_2 that are not in p_1.*

$$\frac{P1, P2 \quad P3 = P1 \circ P2 \quad \varphi \overset{\text{def}}{=} \{ \bigcup_{\substack{p_2 \in P2 \\ p_1 \in P1}} p_3 = (p1.T \cup \{p_2.T \setminus p_1.T\}, p2.S \cup \{p1.S \setminus p2.S\})\}}{P3 = \varphi}$$

With these concepts in mind we can start to define the rules for the valid mission decompositions generation process. In this process we go through the Task Graph in a DFS fashion and we have three possible cases: (i) decomposition nodes, (ii) task nodes and (iii) operator nodes, where the rules for these cases are shown next. Before providing the definitions of these rules let us first define the application of boolean and function effects of a task decomposition. For boolean effects, given some state $s_b \in S_B$ and a task decomposition d, we have:

$$Effect_B(d, s_b) = \begin{cases} s_b, \text{ if } \nexists\ s' \in Eff(d) \mid s_b.pr = s'.pr\ \wedge\ s_b.ps \neq s'.ps \\ s', \ otherwise \end{cases}$$

For function effects, given some state $s_f \in S_F$ and a task decomposition, we have:

$$Effect_F(d, s_f) = \begin{cases} s_f, \text{ if } \nexists\ s' \in Feff(d) \mid s_f.pr = s'.pr \\ s'' = \begin{cases} (s'.pr, s'.val), \text{ if } s'.op = \text{assign} \\ (s'.pr, s_f.val + s'.val), \text{ if } s'.op = increase\ , \ otherwise \\ (s'.pr, s_f.val - s'.val), \text{ if } s'.op = decrease \end{cases} \end{cases}$$

With these two rules in hand, we can proceed to define the effect application function as follows:

$$Effect(d,s) = \begin{cases} Effect_B(d,s), \text{ if } s \in S_B \\ Effect_F(d,s), \text{ if } s \in S_F \end{cases}$$

Given the definitions for application of effects we define the following rule for decomposition nodes.

Rule 13 *Assume that we have the decomposition set D of the Task Graph and a decomposition node $d \in D$. Also, assume that the initial world state of d, $i(d)$, satisfies its preconditions. Thus, the end world state is given by the union of the states generated by the application of d's effects for each $s \in i(d)$.*

$$\frac{d \in D \quad i(d) \vdash Pre(d) \cup Fpre(d) \quad \mu = \bigcup^{s \in i(d)} Effect(d,s)}{e(d) = \mu}$$

Now, we proceed with task nodes. Before providing the rules for this type of node, we need to define the following helper function:

$$\psi(t,d,s) = \begin{cases} p_d = (t \cup \{d\}, e(d)), \text{ if } s \vdash Pre(d) \cup Fpre(d) \\ \emptyset, \text{ otherwise} \end{cases}$$

where t is a set of task decompositions ($\forall a \in t \rightarrow a \in D$), d is a task decomposition ($d \in D$) and s is a state ($s \subseteq S$). Then, we define the next two rules.

Rule 14 *Assume that the valid mission decompositions set P is initially empty. Also, assume that we have a task t in the Task Graph's tasks set T_G and also that we have an initial state i in the set of possible initial states I. Thus, for every decomposition d which is a child of t we create a valid mission decomposition p_d if the initial state i satisfies d's preconditions. This new valid mission decomposition p_d will have its decomposition path composed of d and its world state equal to $e(d)$. In the end, P will be equal to the union of all created valid mission decompositions.*

$$\frac{P = \emptyset \quad t \in T_G \quad i \in I \quad \Phi = \bigcup^{d \in Cn(t)} \psi(\emptyset, d, i)}{P' = \Phi}$$

Rule 15 *Assume that the valid mission decompositions set P is initially not empty and that we have a task t in the Task Graph's tasks set T_G. Thus, for every combination of valid mission decomposition $p \in P$ and decomposition d which is a child of t we create a new valid mission decomposition p_d if and only if the world state of p satisfies d's preconditions. This new valid mission decomposition p_d will have its decomposition path equal to the union of p's decomposition path and d and its world state will be equal to $e(d)$ given as initial state the world state of p. In the end, P will be equal to all of the created valid mission decompositions and the old ones will be dropped.*

$$\frac{P \neq \emptyset \quad t \in T_G \quad \Phi = \bigcup^{\substack{d \in Cn(t) \\ p \in P}} \psi(p.T, d, p.S)}{P' = \Phi}$$

Finally, for operator nodes we have two main cases: (i) AND-decomposed operator node and (ii) OR-decomposed operator nodes. For every AND-decomposed node n_{OP} we have one premise that must hold:

$$\forall\ n_{OP} \in T_{AOP} \implies P'(c) \neq \emptyset,\ \forall c \in Cn(n_{OP})$$

With this in mind we can define the following rules for AND-decomposed nodes.

Rule 16 *Suppose we have an AND-decomposed operator node n_{OP} which is a sequential (;) operator and also that we have nodes $n_1, ..., n_k$ which are the children of n_{OP}. Thus, we have that the final valid mission decompositions set for n_{OP} ($P'(n_{OP})$) is equal to $P'(n_k)$ where for each n_j, $j \in [1, k]$, we have that $P'(n_j)$ is equal to the sequential combination of $P'(n_{j-1})$ and the valid mission decompositions generated by n_j ($P*(n_j)$), given that $n_0 = n_{OP}$.*

$$\frac{n_{OP} \in T_{AOP},\ n_{OP} = \text{";"} \quad k = Cn(n_{OP}) \quad \forall\ i \in [1, k] \implies n_i \in Cn(n_{OP}) \quad n_0 = n_{OP}}{P'(n_{OP}) = P'(n_k) \mid P'(n_j) = P'(n_{j-1}) \circ P^*(n_j), \forall\ n_j \text{ s.t. } j \in [1, k]}$$

Rule 17 *Suppose we have an AND-decomposed operator node n_{OP} which is a parallel (#) operator and also that we have nodes $n_1, ..., n_k$ which are the children of n_{OP}. Thus, we have that the final valid mission decompositions set for n_{OP} ($P'(n_{OP})$) is equal to the parallel combination of all $P'(n_j)$, $j \in [1, k]$, where $P'(n_j)$ is equal to the sequential combination of $P(n_{OP})$ and the valid mission decomposition set generated by n_j ($P*(n_j)$).*

$$\frac{n_{OP} \in T_{AOP},\ n_{OP} = \text{"\#"} \quad k = Cn(n_{OP}) \quad \forall\ i \in [1, k] \implies n_i \in Cn(n_{OP})}{P'(n_{OP}) = \otimes_{j=1}^{k} P'(n_j) \mid P'(n_j) = P(n_{OP}) \circ P^*(n_j), \forall\ n_j}$$

For OR-decomposed nodes, on the other hand, we have one premise that must hold for every node n_{OP}:

$$\exists c \in Cn(n_{OP}) \mid P'(c) \neq \emptyset, \forall\ n_{OP} \in T_{OOP}$$

Since every OR-decomposed operator node must be a parallel one, we have the definition of the rule for this type of node as follows.

Rule 18 *Suppose we have an OR-decomposed operator node n_{OP} and that we have nodes $n_1, ..., n_k$ which are the children of n_{OP}. Thus, we have that the final valid mission decompositions set for n_{OP} ($P'(n_{OP})$) is equal to the union of the final valid mission decomposition sets $P'(n_j)$, $j \in [1, k]$, where each $P'(n_j)$ is equal to the sequential combination of $P(n_{OP})$ and the valid mission decomposition set generated by n_j ($P*(n_j)$).*

$$\frac{n_{OP} \in T_{OOP} \quad k = Cn(n_{OP}) \quad \forall\ i \in [1, k] \implies n_i \in Cn(n_{OP})}{P'(n_{OP}) = \bigcup_{j=1}^{k} P'(n_j) \mid \forall\ n_j \implies P'(n_j) = P(n_{OP}) \circ P^*(n_j)}$$

With all the rules defined we can now provide the arguments for the satisfaction of each of the conditions shown in the beginning of the section.

1. Condition 1: From Rules 14 and 15 we can see that no valid mission decomposition $p \in P$ can contain two decompositions of the same task t in its decomposition path $p.T$
   - By Rule 14 we have that every p_i, $i \in \{1, n\}$ has $p_i.T = \{d_i\}$. It is important to note that n is the number of decompositions of task t which have their preconditions satisfied

- By Rule 15 we have that every p_i, $i \in \{1, n\}$ has $p_i.T = p \cup \{d_i\}$ where $\forall\ a \in p.T \implies !succ(a, t)$ since $p \in P(t)$. It is important to note that n is a combination between the number of current paths in P and decompositions of task t which have their preconditions satisfied

2. Condition 2: From Rules 14 and 15 we can check that every path $p \in P$ will not contain a task decomposition $d \in D$ for which $i(d) \nvdash Pre(d) \cup Fpre(d)$. In order to simplify notation, we call the set of task decompositions of a given task t that had their preconditions satisfied as $D_S(t)$. With this is mind, we have two cases here:

   - If two tasks t_1 and t_2 are AND-decomposed by a sequential operator we will have two valid mission decomposition sets P_1 and P_2 such that $\forall\ d_1 \in D_S(t_1) \implies \exists p \in P_1 \mid d_1 \in p.T$ and $\forall\ d_2 \in D_S(t_2) \implies \exists p \in P_2 \mid d_2 \in p.T$ when the decomposition analyzes the node that corresponds to this sequential operator. Naming the sequential operator node as n_{SEQ}, we have from Rule 16 that these paths will eventually be combined using the sequential combination operator. With this in mind, we have from the definition of the sequential combination operator given in Rule 12 that it generates a new valid mission decomposition set in which every path from P_1 was combined with every path from P_2 by means of a union, thus generating at least one path p for every combination of $d_1 \in D_S(t_1)$ and $d_2 \in D_S(t_2)$.
   - If two tasks t_1 and t_2 are AND-decomposed by a parallel operator we will also have two valid mission decomposition sets P_1 and P_2 such that $\forall\ d_1 \in D_S(t_1) \implies \exists p \in P_1 \mid d_1 \in p.T$ and $\forall\ d_2 \in D_S(t_2) \implies \exists p \in P_2 \mid d_2 \in p.T$ when the decomposition analyzes the node that corresponds to this parallel operator, as is the case for sequentially AND-decomposed tasks. Naming the parallel node as n_{PAR}, we have from Rule 17 that these paths will eventually be combined using the parallel combination operator. First, we have by the parallel combination operator definition in Rule 11 we can see that the resulting valid mission decomposition set generated by it will not contain a path in which two task decompositions conflict. Secondly, we have by the same rule that this new valid mission decomposition set was generated eventually performing the combination of every path from P_1 with every path from P_2 by means of a union, thus generating at least one path p for every combination of $d_1 \in D_S(t_1)$ and $d_2 \in D_S(t_2)$.

3. Condition 3: From Rule 18 we can see that there is no path $p \in P$ for which there are decompositions from two OR-decomposed tasks in the Task Graph present. This is true since for each OR-decomposed node n_{OP} we have that every child generated path $P'(n_i)$ is generated using the initial path $P(n_{OP})$ and since $P'(n_{OP})$ is the union of every $P'(n_i)$. By definition, $P(n_{OP})$ is the set of valid mission decompositions generated before checking node n_{OP} and thus for every task decomposition d which is a child of n_{OP} at some level we have $\nexists\ p \in P \mid d \in p.T$. With this in mind, it is not possible to create new valid mission decomposition for which there are OR-decomposed task decompositions in its decomposition path.

A Additional Functions Explanation

A.1 Constraints Generation Function

The constraints generation function C_{gen} is used to generate constraints for a specific constraint node $ct \in C_N$ of the constraint tree CT. Assuming we have already generated constraints for all child nodes of c_t ($Cn(c_t)$) we generate the constraints for c_t based on them, where the way this is done is shown further for each possible operator.

Sequential Operator Constraint Node If the operator of the current constraint c is a sequential operator we generate constraints between pairs of its children from left to right as shown in Algorithm 1.

For the sequentialConstraintNodeConstraintGen function we have to take into account if the children are constraint or task nodes. This leads us with the following possibilities:

Algorithm 1 Sequential constraint node constraints generation routine

$j \leftarrow 1$
for $i = 0;\ i < c.children.size() - 1;\ i++$ **do**
sequentialConstraintNodeConstraintGen(c.children[i],c.children[j])

j++
end for

(i) If both nodes are tasks we simply generate a sequential constraint between them

(ii) If the first node is a constraint node and the second node is a task we name the first node as c1, the second node as t2 and the parent constraint node as c and have:

A If the first node is a sequential constraint node we have the generation of new constraints as follows:

$$\forall ct \in c1.CT, c.CT = c.CT \cup \{c'\} \text{ where } c' = CTR(ct.N2.TID, t2.TID, SEQ)$$

B If the first node is a fallback or a parallel node we have the generation of new constraints as follows:

$$\forall ct \in c1.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(ct.N1.TID, t2.TID, SEQ) \text{ and}$$
$$c'' = CTR(ct.N2.TID, t2.TID, SEQ)$$

(iii) If the first node is a task and the second node is a constraint node we name the first node as t1, the second node as c2 and the parent constraint node as c and have:

A If the second node is a sequential or fallback constraint node we have the generation of new constraints as follows:

$$\forall ct \in c2.CT,\ c.CT = c.CT \cup \{c'\} \text{ where } c' = CTR(t1.TID, ct.N1.TID, SEQ)$$

B If the second node is a parallel constraint node we have the generation of new constraints as follows:

$$\forall ct \in c1.CT, c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(t1.TID, ct.N1.TID, SEQ) \text{ and}$$
$$c'' = CTR(t1.TID, ct.N2.TID, SEQ)$$

(iv) If both nodes are constraints we name the first node as c1, the second node as c2 and the parent constraint node as c and have:

A If the first node is a sequential constraint node we have:

a If the second node is a sequential or fallback node we have the generation of new constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c'\} \text{ where}$$
$$c' = CTR(ct1.N2.TID, ct2.N1.TID, SEQ)$$

b If the second node is a parallel node we have the generation of new constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where}$$
$$c' = CTR(ct1.N2.TID, ct2.N1.TID, SEQ) \text{ àn}$$
$$c'' = CTR(ct1.N2.TID, ct2.N2.TID, SEQ)$$

B If the first node is a parallel or fallback node we have:

a If the second node is a sequential or fallback node we have the generation of new constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where}$$
$$c' = CTR(ct1.N1.TID, ct2.N1.TID, SEQ) \text{ and}$$
$$c'' = CTR(ct1.N2.TID, ct2.N1.TID, SEQ)$$

b If the second node is a parallel node we have the generation of new constraints as follows:

$$\forall ct1 \in c1.CT,\ ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c'', c''', c''''\} \text{ where}$$
$$c' = CTR(ct1.N1.TID, ct2.N1.TID, SEQ),$$
$$c'' = CTR(ct1.N1.TID, ct2.N2.TID, SEQ),$$
$$c''' = CTR(ct1.N2.TID, ct2.N1.TID, SEQ), \text{ and}$$
$$c'''' = CTR(ct1.N2.TID, ct2.N2.TID, SEQ)$$

If the operator of the current constraint is a fallback operator we generate constraints between pairs of its children from left to right in the following fashion

Fallback Operator Constraint Node If the operator of the current constraint is a fallback operator we generate constraints between pairs of its children from left to right as shown in Algorithm 2.

Algorithm 2 Fallback constraint node constraints generation routine

```
j ← 1
for i = 0; i < c.children.size() − 1; i++ do
    fallbackConstraintNodeConstraintGen(c.children[i],c.children[j])

    j++
end for
```

For the fallbackConstraintNodeConstraintGen function we have to take into account if the children are constraint or task nodes for each pair of child nodes. This leads us with the following possibilities:

(i) If both nodes are tasks we simply generate a fallback constraint between them

(ii) If the first node is a constraint node and the second node is a task we name the first node as c1, the second node as t2 and the parent constraint node as c and have:

A If the first node is a sequential or parallel constraint node we have the generation of constraints as follows:

$$\forall ct \in c1.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(ct.N1.TID, t2.TID, FB) \text{ and}$$
$$c'' = CTR(ct.N2.TID, t2.TID, FB)$$

B If the first node is a fallback constraint node we have the generation of constraints as follows:

$$\forall ct \in c1.CT, c.CT = c.CT \cup \{c'\} \text{ where } c' = CTR(ct.N2.TID, t2.TID, FB)$$

(iii) If the first node is a task and the second node is a constraint node we name the first node as t1, the second node as c2 and the parent constraint node as c and have:

A If the second node is a sequential or parallel constraint node we have the generation of constraints as follows:

$$\forall ct \in c1.CT, c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(t1.TID, ct.N1.TID, FB) \text{ and}$$
$$c'' = CTR(t1.TID, ct.N2.TID, FB)$$

B If the second node is a fallback constraint node we have the generation of constraints as follows:

$$\forall ct \in c2.CT,\ c.CT = c.CT \cup \{c'\} \text{ where } c' = CTR(t1.TID, ct.N1.TID, FB)$$

(iv) If both nodes are constraints we name the first node as c1, the second node as c2 and the parent constraint node as c and have:

A If the first node is a sequential or parallel constraint we have:

a If the second node is a sequential or parallel constraint we have the generation of constraints as follows:

$$\forall ct1 \in c1.CT,\ ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c'', c''', c''''\} \text{ where}$$
$$c' = CTR(ct1.N1.TID, ct2.N1.TID, FB),$$
$$c'' = CTR(ct1.N1.TID, ct2.N2.TID, FB),$$
$$c''' = CTR(ct1.N2.TID, ct2.N1.TID, FB), \text{ and}$$
$$c'''' = CTR(ct1.N2.TID, ct2.N2.TID, FB)$$

b If the second node is a fallback constraint we have the generation of constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where}$$
$$c' = CTR(ct1.N1.TID, ct2.N1.TID, FB) \textit{ and}$$
$$c'' = CTR(ct1.N2.TID, ct2.N1.TID, FB)$$

B If the first node is a fallback constraint we have:

a If the second node is a sequential or parallel constraint we have the generation of constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where}$$
$$c' = CTR(ct1.N2.TID, ct2.N1.TID, FB) \text{ àn}d$$
$$c'' = CTR(ct1.N2.TID, ct2.N2.TID, FB)$$

b If the second node is a fallback constraint we have the generation of constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c'\} \text{ where}$$
$$c' = CTR(ct1.N2.TID, ct2.N1.TID, FB)$$

Parallel Operator Constraint Node If the operator of the current constraint is a sequential operator we generate constraints between all possible pairs of its children from left to right as shown in Algorithm 3.

Algorithm 3 Parallel constraint node constraints generation routine

```
for i = 0; i < c.children.size() − 1; i++ do
    for j = i + 1; j < c.children.size(); j++ do
        parallelConstraintNodeConstraintGen(c.children[i],c.children[j])
    end for
end for
```

For the parallelConstraintNodeConstraintGen function we have to take into account if the children are constraint or task nodes for each pair of child nodes. This leads us with the following possibilities:

(i) If both nodes are tasks we simply generate a parallel constraint between them

(ii) If the first node is a constraint node and the second node is a task we name the first node as c1, the second node as t2 and the parent constraint node as c and have:

A For all types of constraints we have the generation of constraints as follows:

$$\forall ct \in c1.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(ct.N1.TID, t2.TID, PAR) \textit{ and}$$
$$c'' = CTR(ct.N2.TID, t2.TID, PAR)$$

(iii) If the first node is a task and the second node is a constraint node we name the first node as t1, the second node as c2 and the parent constraint node as c and have:

A For all types of constraints we have the generation of constraints as follows:

$$\forall ct \in c1.CT, c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(t1.TID, ct.N1.TID, PAR) \text{ and } c'' = CTR(t1.TID, ct.N2.TID, PAR)$$

(iv) If both nodes are constraints we name the first node as c1, the second node as c2 and the parent constraint node as c and have:

A For all types of constraints we have the generation of constraints as follows:

$$\forall ct1 \in c1.CT,\ ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c'', c''', c''''\} \text{ where } c' = CTR(ct1.N1.TID, ct2.N1.TID, PAR), c'' = CTR(ct1.N1.TID, ct2.N2.TID, PAR), c''' = CTR(ct1.N2.TID, ct2.N1.TID, PAR), \text{ and } c'''' = CTR(ct1.N2.TID, ct2.N2.TID, PAR)$$

A.2 Minimal Constraint Set Function

The minimal constraint set function $MinimalCTR()$ defines the rules for a constraint set C to be considered minimal. The rules for this function depend on the runtime operators involved in the operation and are given next.

For the case where a sequential constraint is trimmed because of a fallback constraint we have:

$$C' : \forall c \in C.SEQ, C' = C \setminus \{c\} \text{ iff } \exists\ c' \in c.SEQ,\ c'' \in c.FB \mid c'.N1 = c.N1, c'.N2 = c''.N1 \text{ and } c''.N2 = c.N2$$

For the case where a sequential constraint is trimmed because of a sequential constraint we have as follows:

$$C' : \forall c \in C.SEQ, C' = C \setminus \{c\} \text{ iff } \exists\ c', c'' \in c.SEQ \mid c'.N1 = c.N1, c'.N2 = c''.N1 \text{ and } c''.N2 = c.N2$$

Finally, for the case where a fallback constraint is trimmed because of a fallback constraint we have:

$$C' : \forall c \in C.FB, C' = C \setminus \{c\} \text{ iff } \exists\ c', c'' \in c.FB \mid c'.N1 = c.N1, c'.N2 = c''.N1 \text{ and } c''.N2 = c.N2$$

B Non-Ground Task Decomposition Graph

In this section we will define a non-ground Task Decomposition Graph (TDG). Inheriting from the definitions given by Bercher et al.[1] we have several (possibly modified) definitions to give:

- Abstract tasks consist of a single element $t(\overline{\tau})$ which consists in a parameterized name

- Primitive tasks (actions) consist of the 3-tuple $(t(\overline{\tau}), pre(\overline{\tau}), eff(\overline{\tau}))$ where $t(\overline{\tau})$ is a parameterized name, $pre(\overline{\tau})$ are preconditions and $eff(\overline{\tau})$ are effects
  - The latter two are conjunctions of literals and depend on the task's parameter variables $\overline{\tau}$
- A partial plan P consists of the 3-tuple $(PS, \prec, VC)$ where PS are the plan steps, ordering constraints $\prec$ and variable constraints VC
  - A plan step $l : t(\overline{\tau}) \in PS$ is a uniquely labeled task
  - The set $\prec$ is a strict partial order on PS
  - The set VC is defined over the variables $\overline{\tau}$ of the tasks of PS
- A method is a tuple $m = (t(\overline{\tau}), P_m, VC_m)$ that maps an abstract task $t(\overline{\tau})$ to its pre-defined partial plan P_m. VC_m denotes a set of variable constraints that relates the variables $\overline{\tau}$ of $t(\overline{\tau})$ with the variables of the partial plan P_m
- A hybrid planning domain is defined by $\mathcal{D} = (T_p, T_a, M)$ which contains the primitive and abstract tasks T_p and T_a, respectively, and a set of methods M
- A hybrid planning problem can be defined as $\mathcal{P} = (\mathcal{D}, P_{init}, C)$ which consists of a domain $\mathcal{D}$, an initial partial plan P_{init} and a set of constants C

With this definition we guarantee that for every method vertex $v_m \in V_m$ we only have variables (or constants) that relate to variables of the root task. Also, since every task vertex $v_t \in V_t \setminus TOP$ belongs to a partial plan of some method vertex, we guarantee that every variable in our task vertices also relate to variables in the root task. In short, every variable in the TDG relates to some variable in the root task, i.e., there is no addition of new variables in any vertex.

With all the definitions we establish the TDG and the decomposition process for each task present in the mission, which is represented by the goal model. Since we use a TDG to decompose a single abstract task $\mathcal{T}$ we always have that the variables in the set of variables $\overline{\tau}$' of the TOP task are exactly the same as the ones defined in the set of variables $\overline{\tau}$ of $\mathcal{T}$ and that the method M which decomposes TOP decomposes it into $\mathcal{T}$. Thus, for each task in the goal model we create the non-ground planning problem $\mathcal{P} = (D, P_{init}, C)$ where we have that $P_{init} = (\{\mathcal{T}(\overline{\tau})\}, \emptyset, \emptyset)$ and:

- For $v_m = M = (TOP(\overline{\tau}'), P_m, VC_m)$ where $P_m = \{\mathcal{T}(\overline{\tau})\}$ and $VC_m = \emptyset$, since $\overline{\tau}' = \overline{\tau}$, we create a single task vertex where it holds:
  - $v_t = \mathcal{T}(\overline{\tau}) \in V_T$
  - $(v_m, v_t) \in E_{M \to T}$

If the same task is present twice in the goal model we have the same result in this process for both instances, since we only verify which task decompositions are valid given a world state in the valid mission decompositions generation step.

Definition 1 *Let $\mathcal{P} = (\mathcal{D}, P_{init}, C)$ be a standard HTN planning problem with domain $\mathcal{D} = (T_p, T_a, M)$. Without loss of generality we assume that P_{init} contains just a single non-ground abstract task TOP for which there is exactly one method in M. The bipartite graph $\mathcal{G} = (V_T, V_M, E_{T \to M}, E_{M \to T})$, consisting of a set of task vertices V_T, method vertices V_M, and edges $E_{T \to M}$ and $E_{M \to T}$ is called the non-ground TDG of $\mathcal{P}$ if it holds:*

1. ***base case*** *(task vertex for the given task)*
   $TOP \in V_T$, the TDG's root.

2. ***method vertices*** *(derived from task vertices)*
   Let $v_t \in V_T$ *with* $v_t = t(\overline{c})$ *and* $(t(\overline{\tau}), P_m, VC_m) \in M$*. Then, with* $v_m = NonGround_{VC_m \cup \{\overline{\tau}=\overline{c}\}}(P_m)$ *as a method vertex holds:*

   - $v_t \in V_T$
   - $(v_t, v_m) \in E_{T \to M}$

3. ***task vertices*** *(derived from method vertices)*
   Let $v_m \in V_m$ *with* $v_m = (PS, \prec, VC)$*. Then, for all plan steps* $l : t(\overline{c}) \in PS$ *with* $v_t = t(\overline{c})$ *holds:*

   - $v_t \in V_T$
   - $(v_m, v_t) \in E_{M \to T}$

4. ***tightness***
   G is minimal, such that 1. to 3. hold

References

[1] Pascal Bercher, Gregor Behnke, Daniel Höller, and Susanne Biundo. An admissible htn planning heuristic. In *IJCAI*, pages 480–488, 2017.