# MASTER'S THESIS

## Model checking Task Models with UPPAAL using MDE

Postma, E.

**Award date:**
2021

# Model checking Task Models with UPPAAL using MDE

Egbert Postma

Student:
Date: 18-06-2021

Open Universiteit
www.ou.nl

# MODEL CHECKING TASK MODELS WITH UPPAAL USING MDE

by

**Egbert Postma**

in partial fulfillment of the requirements for the degree of

**Master of Science**
in Software Engineering

at the Open University of the Netherlands, Faculty of Science,
Master's Programme in Software Engineering
to be defended publicly on Friday July 2, 2021 at 09:00 AM.

Student number:

| | | |
|---|---|---|
| Course code: | IM9906 | |
| Thesis committee: | Dr. Stefano Schivo (chairman), | Open University |
| | Prof. Dr. Tanja Vos (supervisor), | Open University |

# CONTENTS

# SUMMARY

Since the early days of Human-Computer Interaction (HCI) design, tasks and goals have been part of the process. Nowadays, when HCI shifted towards a more digital environment where applications have graphical user interfaces (GUI), the principles of tasks and goals still work. When people use an application, it is mostly because of them wanting to accomplish something, reach a certain goal with it.

Task models are used to write down what interactions are possible with a GUI and show what tasks can be accomplished with a GUI. Task models can also be used for automatic GUI generation. For this to work, task models must be considered *correct*. However, using existing tools, it is very hard to automatically determine whether or not task models are correct. Most tools allow manual simulation of task models, but this is a very tedious and time consuming task. What we have created is a tool that makes it more easy for the user to verify properties of task models and determine if a task model is correct or not.

For the purpose of this study, we have created the TaskTop tool. With TaskTop, users that have knowledge of task models can perform model checking on their task models in a user friendly way. TaskTop allows the user to load in a task model and create queries that verify their correctness.

TaskTop depends on UPPAAL to perform the model checking. UPPAAL is a tool that can execute queries on so called networks of timed automata (NTA). Because task model are not NTA, the task models are first transformed into NTA before they are model checked. This brings us to the main research question that we tried to answer in this thesis: **"How can MDE assist in checking task models with UPPAAL?"**.

For the transformation of the models, we make use of model-driven engineering (MDE). MDE allows using models not only for documentation purposes, but also for the generation of source code or the transformation to models in another domain.

MDE uses metamodels and transformation definitions to transform models from one domain to another. A metamodel can be seen as the blueprint for models and specify the structure to which models must comply. A transformation definition specifies how elements from one metamodel are transformed to elements from another metamodel. This way, we used MDE to transform task models into models that work with UPPAAL.

Experts in modeling task models are, often, not experienced in using UPPAAL. Therefore, in order for our tool to be useful to them, we needed to make sure that TaskTop is accessible to those users. Queries are written in *plain* English and can be constructed using drop-down menus to select the tasks that need verification.

With some simple and more complex use cases, we show that using our tool allows users to verify properties on task models in a simple way. With this, we can proudly say that our tool works and that MDE *can* assist in checking task model with UPPAAL.

# 1

# INTRODUCTION

Since the early days of Human-Computer Interaction (HCI) design, tasks and goals have been part of the process [Card et al., 1983]. Nowadays, when HCI shifted towards a more digital environment where applications have graphical user interfaces (GUI), the principles of tasks and goals still work. When people use an application, it is mostly because of them wanting to accomplish something, reach a certain goal with it.

Take for instance an application with which a user can manage its contacts; a Contact Manager application. The goal of such an application is clear; assist in managing contacts. When one thinks of this applications, a couple of tasks can be described like; open the application, create a new contact, view an existing contact, edit an existing contact (which exists for instance of sub-tasks like; edit the address, edit the phone number, hit the save button.), close application. The user should be enabled to perform these tasks in order to complete the main goal. Therefore, the application should supply the necessary methods that allow the user to fulfill the sub-tasks. This, in turn, means that the application should have, for instance, a button somewhere that creates a new contact for the user and that, after the new contact was created, an editor can be used to enter the new contact's details. After that, when the user has finished entering the contact information, a button for saving the contact is needed. When all sub-tasks are accomplished, the main goal has been achieved and the application has proven to be successful in assisting the user.

The tasks and goals that a user wishes to accomplish can be modeled using so called Task Models [Limbourg and Vanderdonckt, 2003]. In Section 2.1 we will explain what task models are and how they can be used to design graphical user interfaces. It will be clear that there are many different kinds of task models, but we will focus on one formalism that has been widely adopted by GUI designers and developers; ConcurTaskTrees [Paternò et al., 1997].

Task models are mostly used for documentation and reference purposes. However, there exist researches ([Baron and Girard, 2002], [Wolff et al., 2005]) that study automatic GUI generation based on task models. GUI designers and developers use task models to describe how a system or GUI should function based on the tasks. When a task model is finished, the GUI can automatically be generated. The validation of task models is often left behind. Task models are syntactically checked by the GUI generation tool to prevent errors, but semantic checking is not required for such tools. To check if a task model is semantically correct, properties of it should be checked like the reachability of tasks, existence of

deadlocks, safety properties, etc... A tool that is capable of performing such model checking is UPPAAL [Behrmann et al., 2006]. UPPAAL uses timed automata as model inputs. Timed automata are finite state machines that have been enriched with clocks. UPPAAL and can verify all kinds of properties on timed automata. A further explanation of UPPAAL is found in Chapter 2 Section 2.3.

Task models are usually modelled in a different tool than UPPAAL and are not initially modelled as timed automata. Task models can therefore not be tested by UPPAAL immediately. The given task model, which is in the domain of task modellers, should be converted to the UPPAAL domain so UPPAAL is able to perform model checking on it. A good way to convert models between domains is by using MDE (Model-Driven Engineering) [Rodrigues da Silva, 2015]. MDE works with metamodels which can be seen as the blueprint for a certain type of model. A transformation definition between metamodels can be used to transform a model from one domain to the other domain. In Chapter 2 Section 2.4 we will explain MDE further.

In this thesis, we have come up with a tool that allows users to validate task models using UPPAAL without having to *know* UPPAAL. The tool reads a task model file and converts it to a UPPAAL file. The user can then create *user friendly* queries using the tool which are automatically performed on the UPPAAL model. This way, the user does not need to know UPPAAL but can still make use of its powerful model checking capabilities.

In the following Section, we will discuss the *Problem* which we try to overcome by asking ourselves the questions in Section 1.2. Next, we will provide the necessary background information in Chapter 2. Chapter 3 shows what others have already done on subjects that have a strong relation to our subject. Chapter 4 describes the method we have used to perform our research. The validation of our method is found in Chapter 5. Lastly, we will present our conclusion and describe possible future work.

## 1.1. PROBLEM

Task models are commonly used to describe user interfaces. Creators of task models have, however, no way to validate their models, i.e. check that "they make sense". There are tools available like CTTE [Paternò et al., 2001] that allow users and creators of task models to perform simulations on task models. Those tools require the user to manually step through the task model and check if certain scenarios are correct, that is, the execution of tasks at a given time is allowed.

A consequence of manual model verification is that the verification of task models is often skipped or rushed. Skipping or rushing verification can lead to unwanted features that might eventually end up in the user interface (example; a user can perform tasks that should only be possible after the user is logged in).

It would be much more useful if task models could be automatically model checked on certain properties. Properties like reachability (a certain task can eventually be executed) or safety (a user must be logged in in order to execute a certain task) tell a lot about correctness of the task model.

UPPAAL is a tool that can perform model checking on models. However, the models that UPPAAL requires are timed-automata. If task models could be converted into timed-automata, model checking can be performed on task models as well. This way, domain experts who have a deeper knowledge of task models do not need to know UPPAAL at all, but can still benefit of its powerful model checking capabilities.

## 1.2. RESEARCH QUESTIONS

The problems mentioned in the previous section can be tackled with a tool that can perform model checking on task models. As stated in the previous section, task models cannot be directly model checked with UPPAAL, so we will investigate how it is possible to efficiently produce a tool that can perform this task. We will apply an approach based on Model-Driven Engineering (MDE): for this reason, the main research question is;

**Research question** How can MDE assist in checking Task Models with UPPAAL?

We will create a tool that can convert task models into UPPAAL models so we can perform model checking on them. Because we are handling models, we have chosen to look into MDE and see whether it can assist us in doing the conversions. There is another, similar, tool available that does such model conversions to UPPAAL but it works in a different domain, namely that of Attack Trees. To be able to answer our research question, we have subdivided it into three sub-questions;

**Sub-question 1** How do Task Models differ from Attack Trees?

A number of articles are available (i.e. [Kumar et al., 2015, 2018]) in which Attack Trees are described and used to transform into UPPAAL models using the Model-Driven approach of [Schivo et al., 2017]. Because Attack Trees seem in some way similar to Task Models, it might be worthwhile to find similarities, but more importantly, distinguish the differences between them.

**Sub-question 2** How can Task Models be converted to UPPAAL models using MDE?

To be able to check task models using UPPAAL, we need to come up with a transformation between those two models. Using MDE, this can be done by creating a transformation definition which takes the metamodel of task models as input and the metamodel for UPPAAL as output. A task model that conforms to the task metamodel can then be transformed into a UPPAAL model through the transformation engine.

**Sub-question 3** How can property queries be defined in a user friendly way so users do not have to write actual UPPAAL queries?

Because GUI designers and developers usually have no knowledge of UPPAAL, a user friendly way of defining UPPAAL queries should be developed. Users should be able to generate model checking queries in a language or manner they understand.

# 2

# BACKGROUND

In this chapter, we are going to provide background information about the subjects that will be used in the rest of this thesis. We are going to explain what task-models are in Section 2.1. We will give some background information on timed automata and the related UPPAAL tool in Section 2.3. Lastly, we will explain the concept of Model Driven Engineering (MDE) in Section 2.4.

## 2.1. TASK MODELS

A task model is a model that can be used for aiding in the design and development of Graphical User Interfaces (GUIs). There are numerous formalisms in the world of task modelling [Limbourg and Vanderdonckt, 2003]. One of the formalisms that is mostly used is called the ConcurTaskTree (CTT) formalism [Paternò et al., 1997].

A CTT model has a tree-like structure and consists of a root task (a.k.a. goal), (sub)tasks and atomic tasks. In CTT, a task can have two or more sub-tasks if and only if the task is not an atomic task. Sibling tasks are connected through temporal operators (TempOps) which define the temporal relation between the tasks. TempOps among the siblings do not need to be of the same kind. This means that, when a task has three sub-tasks (see Figure 2.1), the TempOp between the first and second sub-task (|||) can be different from the TempOp between the second and third sub-task (>>). Also, a temporal relation only exists between adjacent siblings. A task with three sub-tasks contains only two TempOps, one between the first and second sub-task and one between the second and third sub-task.

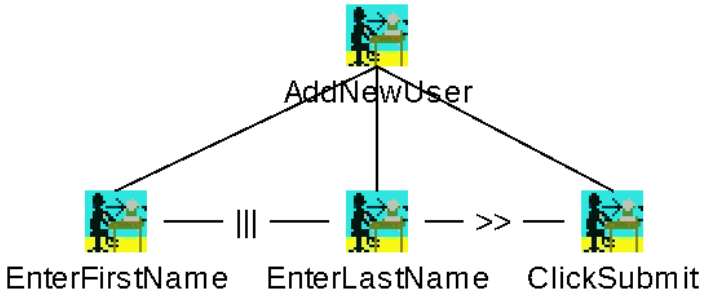



Figure 2.1: CTT example - Add new user task.

## 2.2. TASKS

A task model in CTT is made up of tasks. Tasks represent actions that must be executed in order to fulfill the main goal, the root task. There exist different kinds of tasks which we will explain in Section 2.2.1.

Tasks can be enriched with additional information. CTT allows the use of pre- and post-conditions, as well as time performance values. We will skip the pre- and postconditions for now, as it involves the concept of objects in CTT. Objects are, unfortunately, not very well documented in CTT and the use of it requires more research into how they are meant to be used. Because the use of time performance is quite common for task models, we will include this feature. Section 2.2.1 will explain the use of time performance in more detail.

### 2.2.1. TASK TYPES

**Application task.** Tasks performed by the system are called Application Tasks. The task can provide information for the user of the system, for example, show results to the user. An application task can also perform some internal actions, like validating a provided password. In both cases, the execution of an application task is initiated by the system.

**Interaction task.** Interaction Tasks are tasks initiated by the user and interact with the system. Tasks like 'enter username' or 'click submit' are examples of such interaction tasks.

**User task.** CTT also allows tasks to be User Tasks. User tasks are tasks that do not interact with the system, but are more like tasks that are performed by the user as some kind of decision making. For example, the task 'decide CRUD action' is something the user does in his mind, but does not yet affect the system. A user task is often followed by some choice tasks. In the example, this would be 'select create', 'select read', 'select update' or 'select delete'.

**Abstract task.** If a task has no uniform sub-task types, i.e. an interaction task and an application task, then the parent task must be an abstract task. Atomic tasks cannot be abstract as it has no meaning and may only be used to indicate that a task consists of different types of sub-tasks.

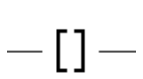
When sub-tasks share the same type, then it is conventional to mark the parent task with the same type. For example, when all sub-tasks are application tasks, then the parent task should also be of type application task.

**Time performance** For every task, a minimum and maximum time can be provided. Setting the time values in CTT is optional. When setting the minimum time, the time to execute a task will take at least the given amount of time-units. The maximum time specifies the maximum duration in time-units of the task to execute. When both the minimum and maximum values are provided, the minimum has to be lower than the maximum. Also, both the minimum and maximum values cannot be negative.

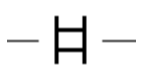
**Definition 1 (Time Performance)** *Time Performance information is defined as TP. Each element of TP is a tuple* $(T_{min}, T_{max})$

### 2.2.2. TEMPORAL OPERATORS

In CTT, sibling tasks are connected through Temporal Operators (TempOps). As mentioned above, a TempOp defines the temporal relation between the tasks. This means that it describes the order in which the tasks can or have to be executed. Also, in some cases, it can make sure that if one task is executed, the other is prohibited from execution (i.e. it becomes disabled). The TempOps that can be used in the CTT notation are based on those defined in the LOTOS specification [ISO 8807:1989]. Also the ordering given by the LOTOS specification is used, which means that one TempOp can have priority over another TempOp. We will now explain the TempOps that can be used in CTT. They are listed in order of priority with highest priority first.

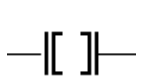


**Choice** The choice operator, as its name suggests, provides the user with a choice between the two tasks. Either the left task is executed or the right task is executed. Initially both the left and the right task are enabled. This means that the user can choose any of the two tasks to execute. When the first is started, the second task becomes disabled and vice versa. In the end, one of the tasks has to be executed in order for the operator to be done.

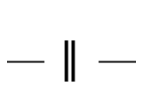


**Order Independence** The Order Independence operator allows the user to execute both tasks in any order, but not at the same time. This means that if, for instance, the first task is started, the second task becomes disabled. This behavior is similar to the choice operator. However, when the first task is done, the second task becomes enabled again and can now be executed. Because the operator is order independent, the second task may also be executed first. In the end, both tasks have to be executed in order for the operator to be done.

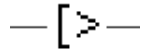
**Interleaving** The Interleaving operator allows the user to execute both tasks in any order and also at the same time. This means that if the first task is started, the second task can also be started at the same time. The operator is also sometimes referred to as the concurrent operator, i.e. the tasks can be executed concurrently. In the end, both tasks have to be executed in order for the operator to be done.



**Synchronization** The Synchronization operator is similar to the Interleaving operator. It also allows the user to execute tasks in any order concurrently. What the Synchronization operator adds is that tasks can exchange information while they are executed. Also for this operator holds that in the end, both tasks have to be executed in order for it to be done.



**Parallel** The Parallel operator is also similar to the Interleaving operator, but it requires both tasks to start at the same time. In many CTT tools, the Parallel operator is not selectable because it is so similar to the Interleaving operator during simulations. Tools that allow the user to select the Parallel operator, like CTTE, interpret the operator as the Interleaving operator during simulation.



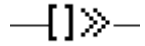
**Disabling** The Disabling operator is used to end (iterative) tasks. When the second task is started, the execution of the first task is ended and it and its sub-tasks become disabled. The operator is considered done when the second task has been executed. An example usage of this operator would be to close a window for instance, or the entire application.



**Suspend/Resume** The Suspend/Resume operator is used to suspend tasks. When the second task is started, the execution of the first task is suspended. When the second task has been executed, the first task is resumed from the state it was suspended in. When the second task has been executed, it becomes enabled again. This means that the first task can be interrupted infinite times. For this operator to be considered done, the first task must have been executed.



**Sequential Enabling** The Sequential Enabling operator ensures that the second task can only be started when the first task has been executed. It implies an order of execution among the tasks. Initially, only the first task will be enabled for execution. In the end, for the operator to be considered done, both the first and the second task must have been executed.



**Sequential Enabling Info** The Sequential Enabling Info operator is generally the same as the regular Sequential Enabling operator. Additionally, information from the first task is to be passed to the second task. An example usage of this operator is, for instance, when the a user enters its username (first task) and submits it for verification (second task). The entered username represents the information that is passed between the tasks.

**Definition 2 (Temporaral Operators)** *The set of TempOps is defined as* $TempOps = \{Choice, OrderIndependence, Interleaving, Synchronization, Parallel, Disabling, SuspendResume, SequentialEnabling, SequentialEnablingInfo\}$ .

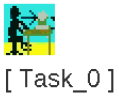
**Definition 3 (Task elements)** *Task elements are defined as* $Elements = TempOps \cup TP$

**Unary operators** Besides the above mentioned binary TempOps, there are also two unary operators possible that can be assigned to tasks themselves.

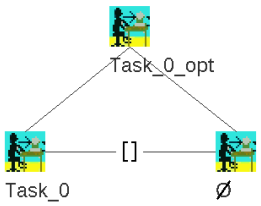
The first operator is the Iterative operator. As the name suggests, it marks the task as iterative, which means it can be executed an infinite amount of times. The iterativeness of a task was already mentioned when we explained the Disabling operator. The disabling operator can be used to stop the iterative task. An iterative task is marked with an asterisk (*).

The other unary operator is the Optional operator. An optional task is not required to be executed, a user can choose to execute the task or skip it. An optional task can therefore also not be used in combination with all binary TempOps. If we look at the Choice operator, it is not possible to mark one of the tasks that belongs to the choice operator as optional. This would not make sense as the entire choice would implicitly become optional. Also the Disabling and Suspend/Resume operators are not possible in combination with an optional task.

As mentioned above, with the optional operator, the user can **choose** to execute the task or skip it. The word 'choose' here implies that we can also replace the optional task with its binary form [Sinnig et al., 2007]:




(a) Optional task - Unary form




(b) Optional task - Binary form

Figure 2.2

In Figure 2.2, $Task\_0$ equals the original optional task and $\emptyset$ equals an empty task. The user can 'choose' between executing the task, or do nothing.

### 2.2.3. CTT VARIANTS

**Prioritized CTT (PCTT).** In CTT it is allowed to have a complex task that exists of multiple sub-tasks that are connected to each other via various kinds of temporal operators. Take for instance the complex task in Figure 2.3. This task can be ambiguous in terms of priorities as the task can be seen as $T := (T0[]T1)|||T2$ or $T := T0[](T1|||T2)$. In the first case, the choice operator has priority over the interleaving operator (Either T0 or T1 have to be executed and T2 has to be executed). In the second case, the interleaving operator has priority over the choice operator (Either T0 has to be executed or T1 and T2 have to be executed). When we apply the standard LOTOS priority order to the example in Figure 2.3, the priority will implicitly be the same as $Task\_0 := Task\_1|||(Task\_2[]Task\_3)|||Task\_4$.

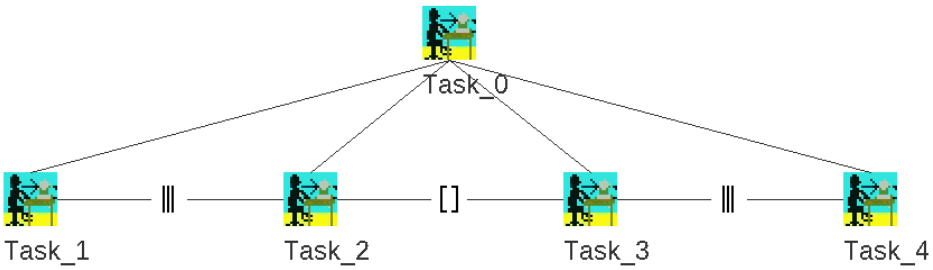



Figure 2.3: A complex task $Task\_0$ with priority issues.

Instead of writing brackets to mark the priorities, it is possible to rewrite the task tree into a priority tree [Paternò et al., 1997]. For ease of use, we will call this a Prioritized CTT or PCTT model. In a priority tree a parent can have multiple children, but the TempOps between the children are the same. The priority tree of the task in Figure 2.3 is shown in Figure 2.4. It can be seen that a new Task node is introduced ($Ta$).

**Definition 4 (PCTT Task model)** *A PCTT Task model TM is a tuple (T, Subtasks, Goal, Iter, Opt, L), where*

- $T$ *is a finite set of tasks.*
- $Subtask : T \rightarrow T^*$ *maps each task to its subtasks.*

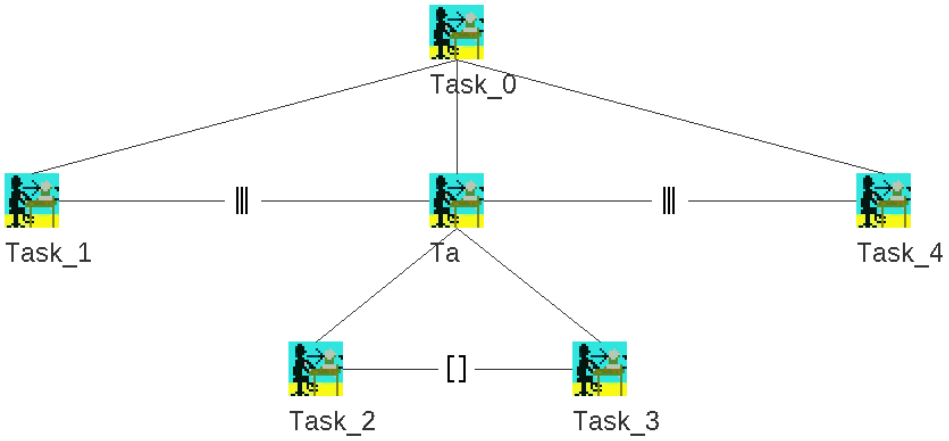



Figure 2.4: Conversion from normal tree to priority tree.

- $Goal \in T$ *is the root task.*
- $Iter : T \rightarrow IsIterative$ *labels each task with an IsIterative value.*
- $Opt : T \rightarrow IsOptional$ *labels each task with an IsOptional value.*
- $L : T \rightarrow Elements$ *labels each task with a Task element.*

A PCTT is well formed when every non-leaf task consists of a minimum of two subtasks; $\mathsf{NonLeaves} = \{v \in V \,|\, |\mathsf{Subtask}(v)| >= 2\}$ and every leaf task of no subtasks, i.e. an empty set; $\mathsf{Leaves} = \{v \in V \,|\, \mathsf{Subtask}(v) = \emptyset\}$. Furthermore, every leaf task is assigned a TP and every non-leaf task a TempOp; $\mathsf{L}(v) \in \mathsf{TP} \leftrightarrow v \in \mathsf{Leaves}$. Also, every task in T should be reachable from the Goal, which is the root item in the tree.

Priority trees can be automatically generated when using the CTTE tool. CTTE will convert the existing task tree into its corresponding priority tree by following the standard LOTOS priority order (i.e. the task tree in Figure 2.3 will be, by default, be converted to the task tree in Figure 2.4). When a different order is required, the user has to explicitly define this.

An advantage of PCTT is that, because all the TempOps among the children are equal, one could say that the TempOps can now be determined by the parent task. That is, the parent task owns the TempOp that is used between its sub-tasks.

**Binary CTT (BCTT).** All task models that are defined in PCTT can also be transformed into their binary equivalents, which we will call Binary CTT or BCTT. A binary task tree contains only tasks with exactly two children when it is not a leaf task. Because of this, a complex task consists of exactly one temporal operator and two sub-tasks. The binary equivalent of the example in Figure 2.4 can be found in Figure 2.5. The binary tree differs from the priority tree in such a way that all tasks that have more than two subtasks are split. The first subtask stays in place, where the rest of the subtasks (also known as the body of subtasks) is cut and replaced by a single new subtask under which the body is placed. This is repeated until all tasks have exactly two subtasks. In the example in Figure 2.3, the first subtask ($Task\_1$) stays in place and the body ($\{Ta, Task\_4\}$) is cut and replaced by a new task ($Tb$) under which the body is pasted.

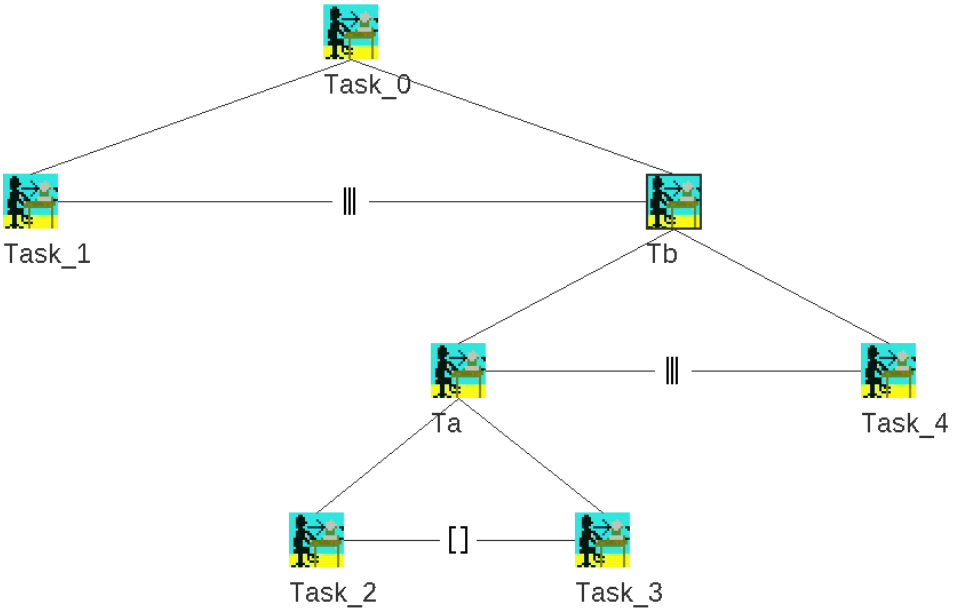



Figure 2.5: Binary task tree

### 2.2.4. Enabled Task Sets

Enabled Task Sets (ETSs) are a part of task models. As specified by [Paternò, 2000], ETSs are sets of tasks that are enabled at any given state of the task model. Initially, a task model has a one or more *enabled tasks*. When one of that tasks is executed, the state of the task model changes to the next. In this next state, the executed task might be disabled and new tasks may have been enabled. The tasks that are enabled in this next state represent the next ETS. ETSs can be modelled by a simple State Transition Network (STN) like the one seen in Figure 2.7, which represents the ETSs of the task model in Figure 2.6.

When the task model of Figure 2.6 is initialized, the first tasks that are enabled are *Task_3* and *Task_4*. If *Task_3* is executed, the enabled task becomes *Task_4*. A choice was made to execute the left task, *Task_1* of the root task, *Task_0*, which disables *Task_2* and all of its subtasks (*Task_5* and *Task_6*). When *Task_4* is now executed, the task model is done and the main goal, *Task_0* is executed. Instead, if *Task_5* had been executed in the first ETS, then all of the subtasks of *Task_1* would have been disabled.

How one gets the ETSs of a certain CTT task model is also described in [Paternò, 2000]. ETSs can be useful when, for example, creating user interfaces. Every ETS then represents a curent set of available actions on the display. When the user, for instance, clicks a button, the UI goes to its next state. This next state also corresponds to an ETS and, therefore, has its own available actions. To generate ETSs, one can use the CTTE tool. Using CTTE, a user can create task models and generate ETSs for that task model as well.

### 2.2.5. Task states

In the previous Section, we have discussed ETSs. From that, we have learned that tasks can be enabled or disabled and that tasks can be executed. That this is possible leads us to the fact that the tasks, when run in a task model, can exist in different states at any given time. When we, for example, want to simulate a task model, all tasks are disabled. This must be the case, because whether or not a task is enabled depends on the temporal operator it is

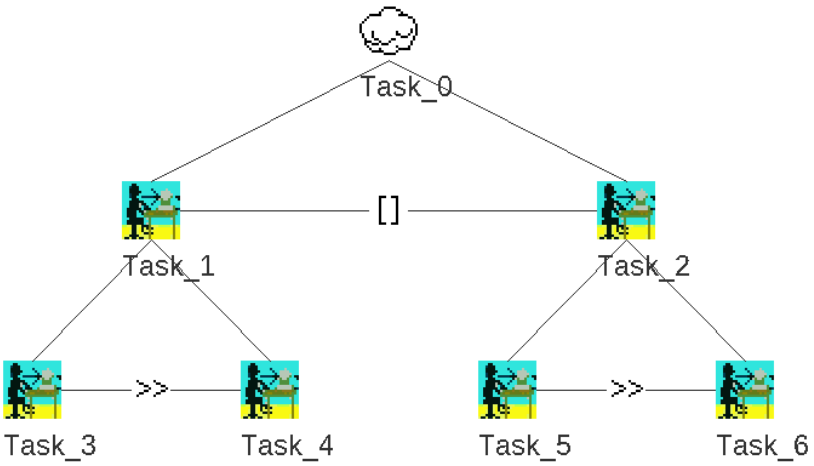



Figure 2.6: A simple task model to demonstrate ETSs.

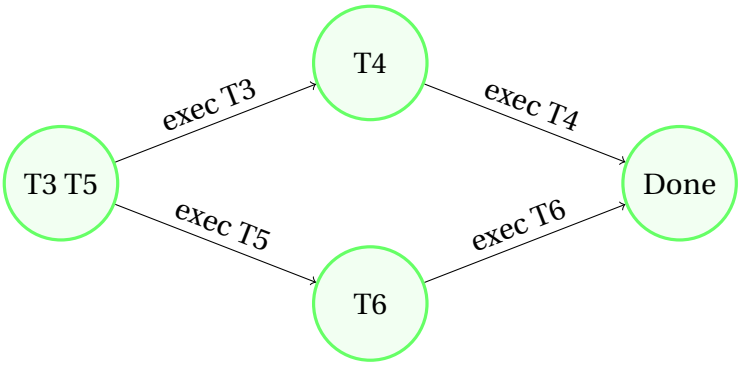



Figure 2.7: The ETS state model for Figure 2.6

connected to. One exception to this is the root task, which is enabled right from the start.

**Leaf task state model** Because every non-leaf task is connected to a temporal operator; i.e. it acts as the operator, it determines the initial state of its subtasks. The root task is the first non-leaf task and enables its subtasks based on the operator. A subtask that is enabled, will also enable its subtasks based on its operator. This continues until the leaf tasks are reached. Which tasks become enabled are the same as the ones that would have been in the initial ETS, as mentioned in the previous section.

Based on the states a task goes through, a state model can be described. A non-leaf task (which has an operator) follows a slightly different state model than a leaf task but in general the state model is as it is shown in Figure 2.8. As mentioned before, a task starts in the *Disabled* state. A task, that is enabled by its parent, moves to the *Enabled* state and waits until it is started.

When a task is started, it goes to the *Active* state. Tasks of type *interaction* will, in general, be started by the user. Application tasks are, however, started automatically. Tasks can have minimum and maximum time parameters set. When a minimum time is set, the task is, at least, active for that time before it can transition to the *Done* state. When the maximum time is set, it can stay in the *Active* state for that given time. When both minimum time and maximum time are set, the task has to stay in the *Active* state for the given time span.

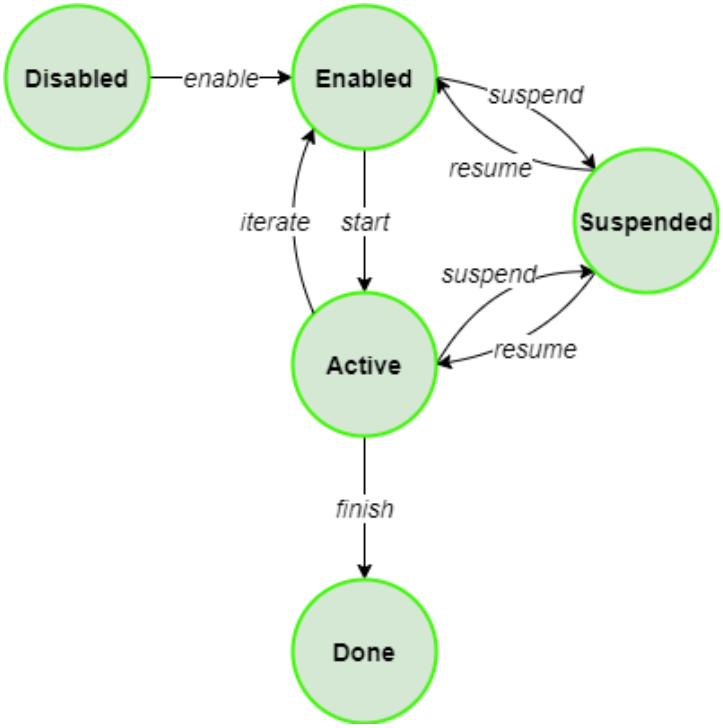



Figure 2.8: CTT state model

When a task is finished, it moves to the *Done* state. Here, the task will wait indefinitely. In case the task is marked as **iterative**, it will never become done. This means that when an iterative task is finished, it moves to the *Enabled* state again so it can be restarted. A task can be reset to the *Disabled* state at any time by the parent. This means that it can transition from every state to the *Disabled* state. This is, however, not shown in Figure 2.8 to keep the model clear.

The state model also includes the *Suspended* state. The task can transition to this state from the *Enabled* state and *Active* state. The *Suspended* state is used when the task is part of a parent task that has a *SuspendResume* operator. Important is that it can only transition back to the state it came frome. Thus, when it moves to the *Suspended* state when it was in the *Enabled* state, it has to transition back to the *Enabled* state.

**Non-leaf task state model** The state model for the non-leaf tasks follow, in general, the same task model as leaf tasks (see Figure 2.8). The difference is that non-leaf tasks determine when their subtasks are enabled based on the operator that is assigned to it. For instance, a choice operator will initially enable both subtasks of the non-leaf task whereas a *SequentialEnabling* operator will initially only enable the left subtask.

Whether a task in the task model is enabled or disabled is determined from top to bottom. When a non-leaf task is disabled, all of its subtasks are disabled too. Note that when a non-leaf task is enabled, this does **not** imply that all of its subtasks become enabled as this depends on the operators of the non-leaf subtasks. In the example of Figure 2.9, the choice was made to execute Task_3. This resulted in Task_2 being disabled. This, in turn, results in the entire sub-branch of Task_2 being disabled. Note that it is not visible in a CTT model that a task is being executed. In Figure 2.9, it is assumed that one of the subtasks of Task_3 is being executed.

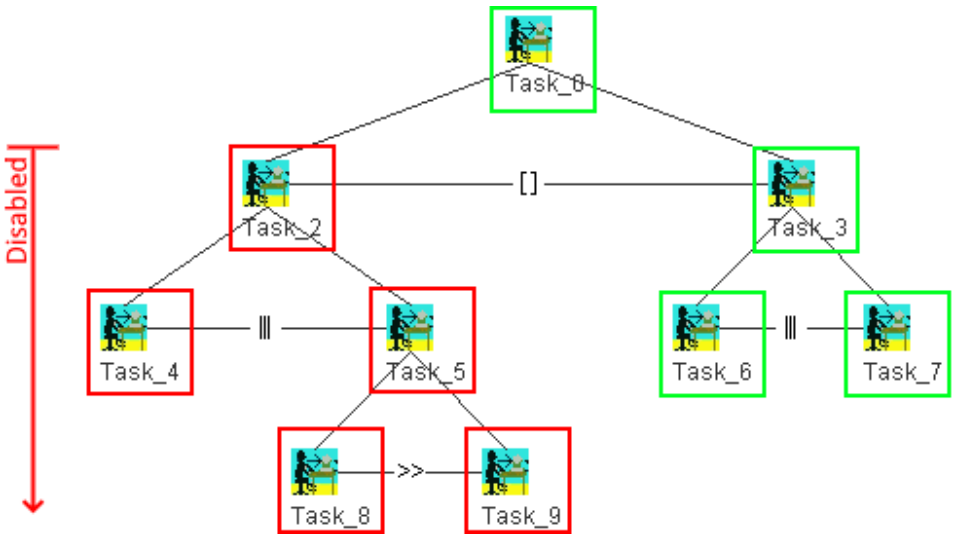



Figure 2.9: When Task_3 is being executed, Task_2 is automatically disabled because of the *choice* operator. The red squares indicate disabled tasks and the green squares indicate enabled tasks. Whether or not a task is active (or being executed) is not visible in this diagram.

When a leaf-task is started, it notifies its parent that it has started. The non-leaf task (parent) then notifies its parent until it reaches an already started non-leaf task or the root task. For example, a non-leaf task with a choice operator, needs to know when one of its subtasks is started. It then immediately disables the other subtask and also notifies its parent task that it has been started.

The same yields for tasks being that become done. When a task becomes done, it notifies its parent task that it is done. The parent task will act based on the operator that is assigned to it. If the parent task has, for instance, a *SequentialEnabling* operator and the task that became done was the left subtask, it will enable the right subtask. If, however, the parent task has a *Choice* operator, it will also become done and notify its parent (i.e. a choice task needs only one subtask to be done for it to become done).

Whether a task in the task model is started or done is determined from bottom to top. When a leaf task is becomes active, all the parent tasks up in that branch become active too. Note that when a leaf task becomes done, this does **not** imply that all of its parent tasks become done as this depends on the operators of the parent tasks. In Figure 2.10, the user decided to start Task_8. Because of this, its parent task (Task_5) is also started and so is its parent task, all the way up to the root task (Task_0). It can happen that a parent task has already been started by one of its other subtasks. In this case, the parent task and all of the tasks above it in the same branch do not need to be started again. If, for instance, Task_4 would also be started, it notifies Task_2 that it has started, but then it stops because Task_2 was already active.

Suspending and resuming tasks works quite like disabling tasks. When a non-leaf task is suspended, all of its subtasks are suspended too. The same yields for resuming tasks. When the non-leaf task is resumed, all of its subtasks are resumed. An example is shown in Figure 2.11. When Task_3 is started, Task_2 and all of its subtasks are suspended. The same yields for resuming. When Task_3 is done, Task_2 and all of its subtasks are resumed. Task_3 is automatically re-enabled by Task_0 so it can be executed again as this is part of how *SuspendResume* operators work.

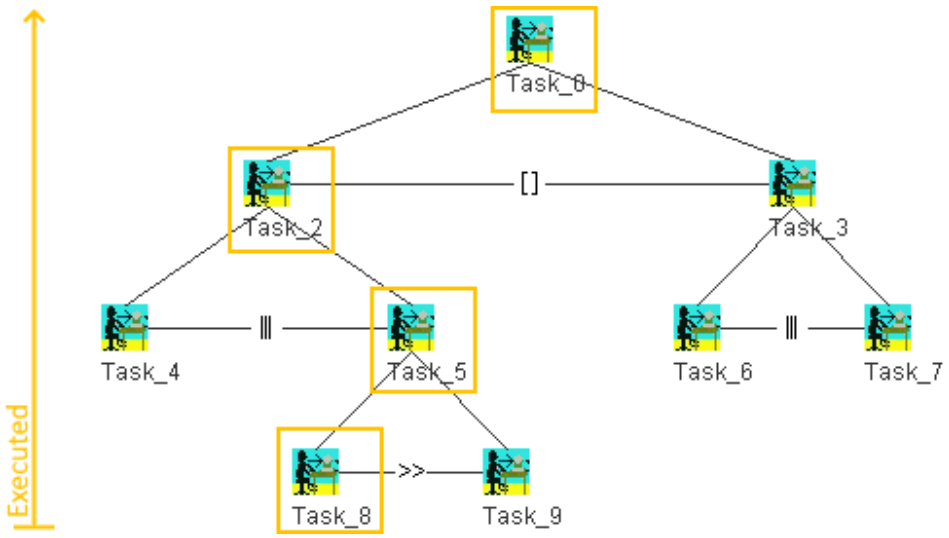



Figure 2.10: An example of a started task notifying its parent.

When a task is suspended, it remembers the state that it came from, i.e. Enabled or Active. When the task is later resumed, it goes back to that state. Tasks that are disabled or done cannot be suspended as they are in a 'stable' state. Stable means in this case that a task cannot perform actions by itself.

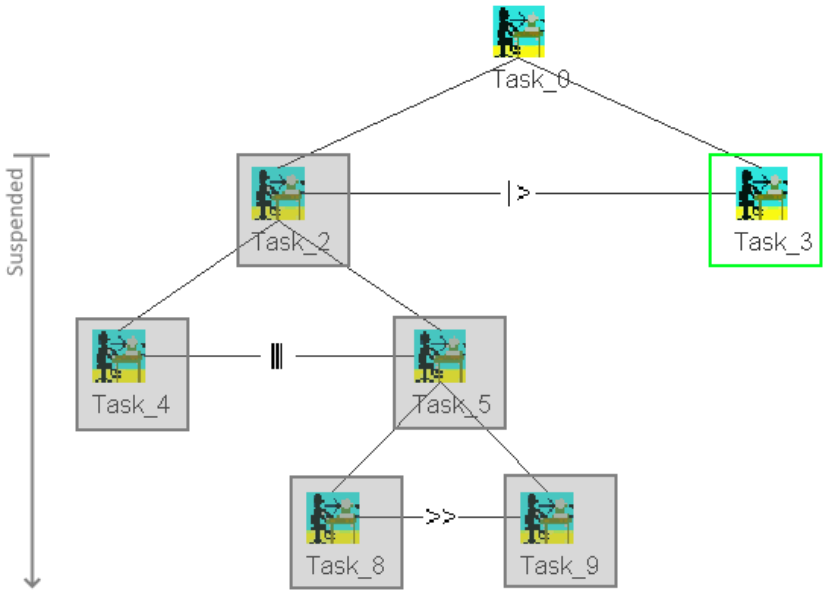



Figure 2.11: An example of a suspended task. The entire branch of the suspended task is also suspended.

## 2.3. UPPAAL

UPPAAL is a tool that is used to analyze timed automata. Timed automata are finite state machines (or simply state machines) enriched with a finite set of clocks [Alur and Dill, 1992]. With timed automata it is possible to put clock constraints to states and transitions. Clocks can also be individually reset on a transition in the model.

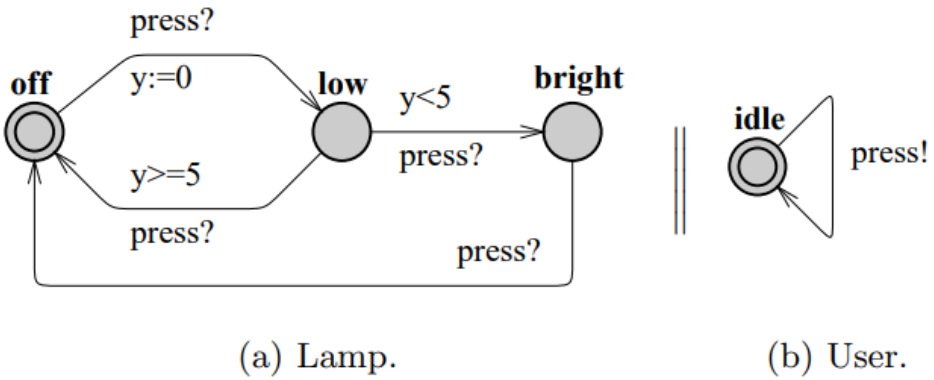



Figure 2.12: An example of a timed automaton in UPPAAL (taken from [Behrmann et al., 2004]).

The example in figure 2.12 shows a model of a lamp and a button. The initial state of both the lamp and the user are `off` and `idle` respectively (annotated with a double ring). When the user presses the button, `press!` is fired to which the lamp automaton transitions to the `low` state. In this transition, clock `y` is reset to 0. If the user presses the button again within 5 clock ticks, the lamp automaton transitions to the `bright` state. Else, if the user does nothing, the lamp automaton stays in the `low` state. If the user eventually presses the button again (thus after at least 5 clock ticks), the lamp model returns to the `off` state. When, however, the lamp automaton was in the `bright` state, and the user pressed the button, the lamp automaton also returns to the `off` state.

### 2.3.1. Notation

In the example of Figure 2.12, the basics of a network of timed automata (NTA) is shown. The NTA in the example consists of two *Templates*; the Lamp template and the User template.

#### Templates

A template in UPPAAL can be seen as the blueprint for a timed automaton. A template consists of *Locations* and *Edges*, where *Locations* represent the states of the timed automaton and the *Edges* represent the transitions between the states. A template is instantiated and assigned to the system as a process in the *System declarations* section of the NTA. Besides the system declarations, there are also other declarations like channel, clock and variable declarations.

Templates can be given input parameters that are used to configure templates or pass information to them. Parameters can be passed by value as well as by reference. The parameters are set when the template is instantiated in the system declarations section of the NTA. An example of a system declaration in UPPAAL is given in Listing 1. In the example, a template called *TopLevel* is instantiated with a parameter 0 and is named `top_level` so it can be referenced later. On the last line of this listing, the created instance is assigned to the *system* so it is available to the model checker.

#### Locations

*Locations* are displayed as nodes (see Figure 2.13a) and have a unique name in the template. To a location, one can assign an *invariant*. An invariant must evaluate to true in order for the timed automaton to be in that location. The invariant `x < 5` means that the

current state of the timed automaton can be in the given location as long as $x$ is smaller than 5. A special kind of invariant is the *clock rate* invariant. A clock rate invariant sets the rate of the clock when the current location is active. In normal UPPAAL (the version without the Statistical Model-Checker, or SMC), the clock rate can be set to either 0 or 1 (`x' == 0` or `x' == 1`) where 0 means that the clock has stopped and 1 means that the clock is running at its default rate. Multiple invariants can be set using the *and* (`&&`) and *or* (||) operators. When combining a normal invariant and a clock rate invariant, one must use the `&&` operator and assign the clock rate invariant as the latter one.

Furthermore, a location can be marked as initial, urgent and/or committed. An initial location is the location in which the automaton will start. Note that only one location in the template can be marked as initial. An initial task is recognised by its double rings (Figure 2.13b). Locations that are marked as urgent or committed do not allow time to pass, i.e. the locations freeze time. When a process is in an urgent location, other processes can let time pass before the urgent location is left. When, however, a process is in a committed location, an outgoing edge from the committed location must be taken first before other processes can proceed. Urgent locations are marked with a 'U' (Figure 2.13c) and committed locations with a 'C' (Figure 2.13d).

(a) Normal location (b) Initial location (c) Urgent location (d) Committed location

Figure 2.13: Locations in UPPAAL

### EDGES

*Edges* are shown as the lines between nodes. *Edges* have a direction (arrow) and represent a transition between nodes in that direction. An *Edge* can, like *Locations*, also have extra information assigned to them, namely; guards, synchronizations, updates and selections.

**Guards** A *guard* statement is used to 'guard' the transition. This means that the guard expression must be satisfied in order to allow the transition to happen. The guard `x>2` means that the transition may only be taken when the value of $x$ is greater than 5.

**Synchronizations** A *synchronization* statement is used to either send or receive synchronizations to or from other templates through channels that are declared in the global declarations of the NTA. In the example of the lamp in Figure 2.12, `press` is such a channel synchronization. On the edges in the lamp template, the channel is used as a receiving synchronization as it the name of the channel is followed by a `?`. On the edge in the user template, the channel name is followed by an `!` which indicates that it sends a synchronization on the channel. A sending synchronization must be completed with a receiving synchronization. If there is no receiving synchronization available on a transition in another template instance, the model is locked. If there are multiple receiving synchronizations, only one can be synchronized.f Note that there is an exception to this rule, which is when the channel is a broadcast channel. a broadcast channel may be listened to by zero to many receiving synchronizations.

**Updates** An *update* statement can be used to update variables in the system. It can be used to reset clocks (`x = 0`) or, for instance, to assign a value to a boolean variable (`selected = true`). Multiple update expressions can be set when separated using a comma, as in `x = 0, i = 5`.

**Selections** Although a *select* is the first item one can set on a transition, we cover it last here because it can be used in the guards, synchronizations and updates. A selection is used to introduce temporary variables for that transition only. A value will be non-deterministically assigned to the variable from all possible values. The selection `i: int` will, therefore, assign a random value between -32768 and 32767 to the variable `i` as that are the bounds of an integer. Note that, when using a regular int, the number of possible transitions is enormous. This can be prevented by using a bounded integer instead, like `i: int[0,2]`, which will only assign either a 0, 1 or 2 to `i`.

### 2.3.2. PROPERTY VERIFICATION

UPPAAL can verify timed automata for the following properties [Behrmann et al., 2004]:

- **liveness**, which indicates that some state will eventually be reached.
- **safety**, which indicates that some state will never occur.
- **reachability**, which indicates that some state can occur.

Queries in UPPAAL are based on Timed Computation Tree Logic (TCTL) [Behrmann et al., 2004]. In the example given in figure 2.12, the reachability of state `bright` could be verified using UPPAAL using the following query:

$$E <> bright \tag{2.1}$$

which in short says something like: "State `bright` exists ($E$) eventually (<>)." Figure 2.14 shows the possible TCTL formulas in UPPAAL.

**Liveness properties** As written before, liveness properties indicate that some state will eventually happen. In the TCTL notation, liveness properties can be written using **A<>** $\varphi$ or $\varphi \rightsquigarrow \psi$. The first means that $\varphi$ will eventually be satisfied, whilst the latter means that, once $\varphi$ is satisfied, then eventually $\psi$ will be satisfied.

**Safety properties** A safety property is used to indicate that some state will never occur, or on the contrary, that a state always occurs. In UPPAAL, both **A[]** and **E[]** are used to define safety properties. **A[]** $\varphi$ is satisfied when, in every possible path, $\varphi$ is $true$. When $\varphi$ means "not error", then **A[]** $\varphi$ means, that a state in which "error" yields can never be reached. **E[]** $\varphi$ means that there exists at least one path in which $\varphi$ is always true.

**Reachabillity properties** Reachability properties are used to validate that a certain state is possible, i.e. there exists a path to a state in which the property is true. In UPPAAL, we can use **E<>** to check for reachability. **E<>** $\varphi$ means that there exists a path in which $\varphi$ is true.

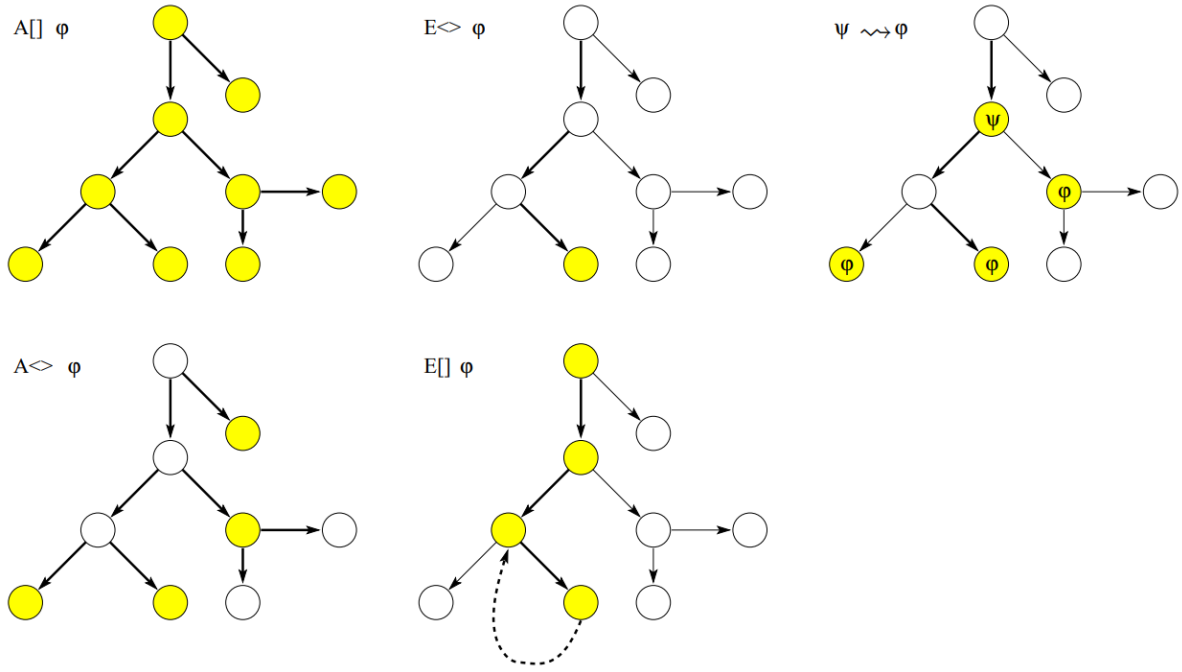



Figure 2.14: Examples of path formulas, taken from [Behrmann et al., 2004]

### TRACES

When UPPAAL checks a property, it can be satisfied or not satisfied. Also, UPPAAL outputs the trace that shows how a property was satisfied (evidence) or the trace that shows how a property could not be satisfied (counter-example). A trace consists of all the states the model has been in and all the transitions that have been taken to get to the final state. UPPAAL allows the user to import the generated trace in the simulator so the user can replay the trace. UPPAAL can look for various types of traces that lead to the property, like for instance:

- the shortest trace (least amount of transitions);
- fastest trace (least amount of clock steps);
- some trace (first trace found)

A special version of UPPAAL, UPPAAL CORA [Behrmann et al., 2005], introduces the `cost` variable to timed automata and creates priced timed automata (PTA). The `cost` variable can be set with an update on a transition using `cost' += <val>`. Also, the `cost'` variable can be increased over time with a given rate which can be set specifying a locations invariant as `cost' == <rate>`. UPPAAL CORA tries to find the 'cheapest' trace (with the lowest cost) that leads to the given property and so extends the capabilities of the regular UPPAAL tool.

## 2.4. MODEL-DRIVEN ENGINEERING

Often, models are used to describe the concepts and behaviour of certain domains. In many cases, models are found in the documentation of such a domain. For instance, in the GUI design domain, task models are used to describe what actions can be performed and

how different items (widgets) of the GUI map to those actions. In model driven engineering, or MDE, models are not only used for documentation, but are also used to assist in the development of GUIs.

For developers to be able to use models for development, the models need to be specified in a formalized way and need to conform to some sort of blueprint. Every domain, like the task model domain, can be described in such a blueprint. In MDE, those blueprints are called metamodels and describe the structure of a model [Rodrigues da Silva, 2015]. A model can thus be seen as an instance of a metamodel. A metamodel consists of the core concepts of the domain. When we take the task domain for instance, every model at least consist of 'Tasks' and 'Operators'. These concepts characterize a task model.

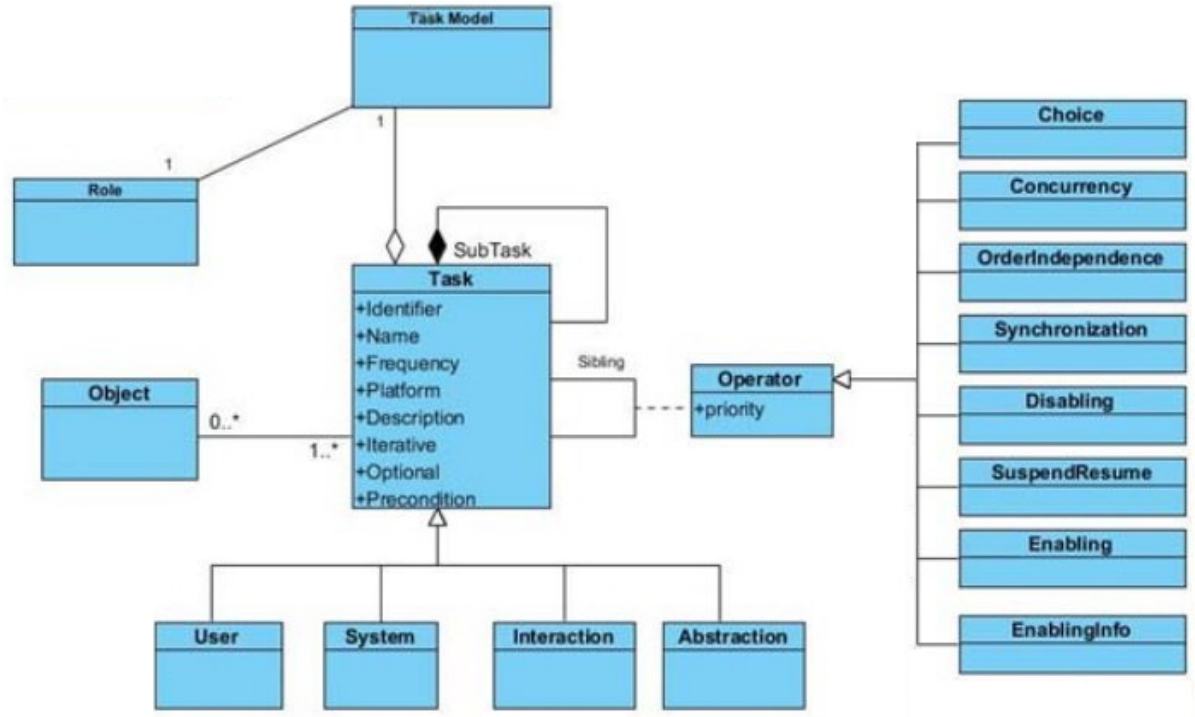



Figure 2.15: A metamodel for task models based on CTT, taken from https://www.w3.org/2012/02/ctt/ and was slightly cleaned up by removing non-used classes.

Figure 2.15 shows a metamodel for the task models. This particular metamodel is based on CTT. The concept of *Task* that lives in the metamodel is specified as super class and is sub-classed by *UserTask*, *SystemTask*, *InteractionTask* and *AbstractTask*. Every *Task* has a list of *Objects* that it has access to and can therefore be modified by the *Task*. *Tasks* can have *Subtasks* and sibling *Subtasks* are connected with each other through *Operators*. A *Task* can marked *Optional* and/or *Iterative* by setting its boolean values accordingly. Besides the *Optional* and *Iterative* properties, *Tasks* can also be configured with *Preconditions* and *Postconditions*.

**Model transformations** Model Driven Engineering also allows the transformation of models. The diagram in Figure 2.16 shows an overview of how transformations work. The source model, which conforms to the source metamodel, can be transformed to the target model, which conforms to the target metamodel. A transformation definition maps elements of

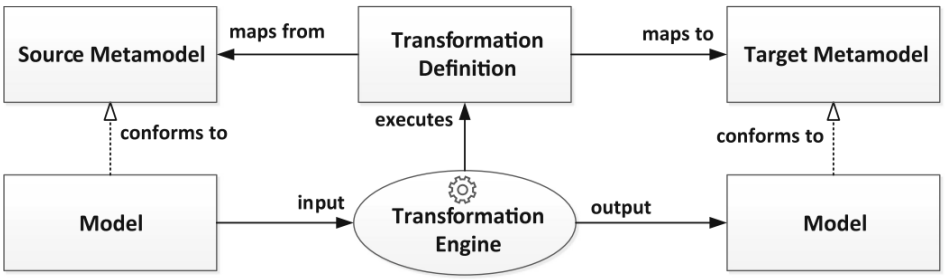



Figure 2.16: An overview of how model transformations work, taken from [Schivo et al., 2017]

the source metamodel to elements in the target metamodel. A so called transformation engine is able to perform the transformations and converts the actual models. This way, in the end, users can create models in their domain of knowledge, which, in turn can be used in another domain.

**Eclipse Modeling Framework** The Eclipse Modeling Framework (EMF) is a framework that was developed by the Eclipse Foundation [1]. The framework is widely used to assist in projects that use MDE. The framework supports the creation of metamodels using its own metamodel format ECore. An ECore file is written in the XMI notation. With ECore, one can define their own metamodel.

The EMF framework is also capable of generating Java code from the models created with the ECore metamodel. A complete set of Java files, including interface and implementation classes, factory classes and package classes, are generated by the framework so the models can be easily used in, for instance, a Java application.

An addition to the EMF framework is the Epsilon framework. Epsilon provides Domain Specific Languages to create metamodels and models. Epsilon works with UML, XML, Simulink and also EMF. Ecore metamodel can be created in a more simple manner using Epsilon. An example of a metamodel defined with the Epsilon framework can be found in Figure 2.17a. The example represents the metamodel of a simple Graph. As one can see, the definition is fairly simple and clear to understand. From the Epsilon Emfatic file, the ECore file in Figure 2.17b is generated.

The Epsilon framework also contains the ETL notation for writing model-to-model transformations. With ETL, one can write scripts to transform models conforming to metamodel *A* to models that conform to metamodel *B* (*Note that A and B can be the same metamodel!*. In Figure 2.18b, an example transformation is given for transforming a Tree model in to a Graph model. The transformation script consists of *rules*. Every rule is considered and looks for the elements specified after the *transform* keyword (in the example this is the Tree element of the Tree model, see Figure 2.18a). For every *Tree* element, a *Node* element (from the Graph metamodels) will be created. With that, the *name* property of the *Node* element will be set to the value of the *label* property of the *Tree* element. At last, if the *Tree* element has a parent, an *Edge* element (from the Graph metamodels) will be created of which the *source* and *target* properties will be set accordingly.

[1] https://www.eclipse.org/modeling/emf/

```
package Graph;

class Graph {
    val Node[*] nodes;
}

class Node {
    attr String name;
    val Edge[*]#source outgoing;
    ref Edge[*]#target incoming;
}

class Edge {
    ref Node#outgoing source;
    ref Node#incoming target;
}
```

(a) Emfatice format

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="Graph"
    nsURI="Graph" nsPrefix="Graph">
  <eClassifiers xsi:type="ecore:EClass" name="Graph">
    <eStructuralFeatures xsi:type="ecore:EReference" name="nodes" upperBound="-1"
        eType="#//Node" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Node">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="outgoing" upperBound="-1"
        eType="#//Edge" containment="true" eOpposite="#//Edge/source"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="incoming" upperBound="-1"
        eType="#//Edge" eOpposite="#//Edge/target"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Edge">
    <eStructuralFeatures xsi:type="ecore:EReference" name="source" eType="#//Node"
        eOpposite="#//Node/outgoing"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="target" eType="#//Node"
        eOpposite="#//Node/incoming"/>
  </eClassifiers>
</ecore:EPackage>
```

(b) Ecore format

Figure 2.17: A comparison between the Emfatic notation and the ECore notation. The Emfatic notation of the metamodel is much more readable than the ECore notation.

```
package Tree;

class Tree {
    val Tree[*]#parent children;
    ref Tree#children parent;
    attr String label;
}
```

(a) Tree metamodel, defined in Emfatic.

```
rule Tree2Node
    transform t : Tree!Tree
    to n : Graph!Node {

    n.name = t.label;

    // If t is not the top tree
    // create an edge connecting n
    // with the Node created from t's parent
    if (t.parent.isDefined()) {
        var e : new Graph!Edge;
        e.source ::= t.parent;
        e.target = n;
    }
}
```

(b) ETL example

Figure 2.18: On the left, the metamodel for a Tree structure model. On the right a transformation that transforms a Tree model into a Graph.

# 3

# RELATED WORK

### ABSTRACT SYNTAX OF CTT

CTT was standardized in [Paternò et al., 2014] but there is no clear abstract syntax provided. An abstract syntax was described for the Extended CTT notation formalism [Sinnig et al., 2011]. ECTT is a version of CTT extended with two extra temporal operators; Stop and Resume. Also, ECTT allows multiple root goal nodes that can be referenced to by leaf tasks. ECTT therefore allows users to create modular task models and therefore does not require the model to be monolithic. The original CTT, however, does not allow this and requires the task models to be one defined as a whole. When the extra operators are left out of the abstract syntax, and do not allow multiple root goal nodes, we are left with the notation for the CTT formalism.

### CTT ENVIRONMENT TOOL

To create CTT task models, the CTTE (ConcurTaskTree Environment) tool was created [Paternò et al., 2001]. The tool allows users to not only create CTT task models, but also analyze and simulate them. CTTE works with the original CTT notation as standardized by [Paternò et al., 2014]. The task models that are created are thus monolithic models and can not be split up into multiple models. The tool does, however, also work with the cooperative version of CTT. The cooperative version of CTT allows the user to model multiple goals, but also the interaction between those models. Take for instance a task model that describes the landing of an airplane and a task model that describes the handling of incoming airplanes on a runway. The cooperative version of CTT can be used to model the interaction between those two task models. CTTE can also analyze and simulate cooperative task models.

To maintain compatibility, we use the standardized version of CTT instead of the ECTT. This way, we can use the CTTE tool to create CTT task models and use the output files of that tool to convert to UPPAAL models for model checking.

### ATTACK TREES

Attack Trees [Mauw and Oostdijk, 2006; Schneier, 1999] are graphical representations of how a system could be attacked. An overview of different types of attack trees can be found in [Kordy et al., 2014]. The nodes in an attack tree form the attacks (or counter measures in some special attack trees [Kordy et al., 2011] called Attack-Defense Trees) which can be refined into sub-attacks. How the sub-attacks must be executed is denoted by so called

gates, of which every non-leaf attack corresponds to. Leaf-attacks are attacks that have no sub-attacks anymore. These are the actual attacks that can be executed by an attacker.

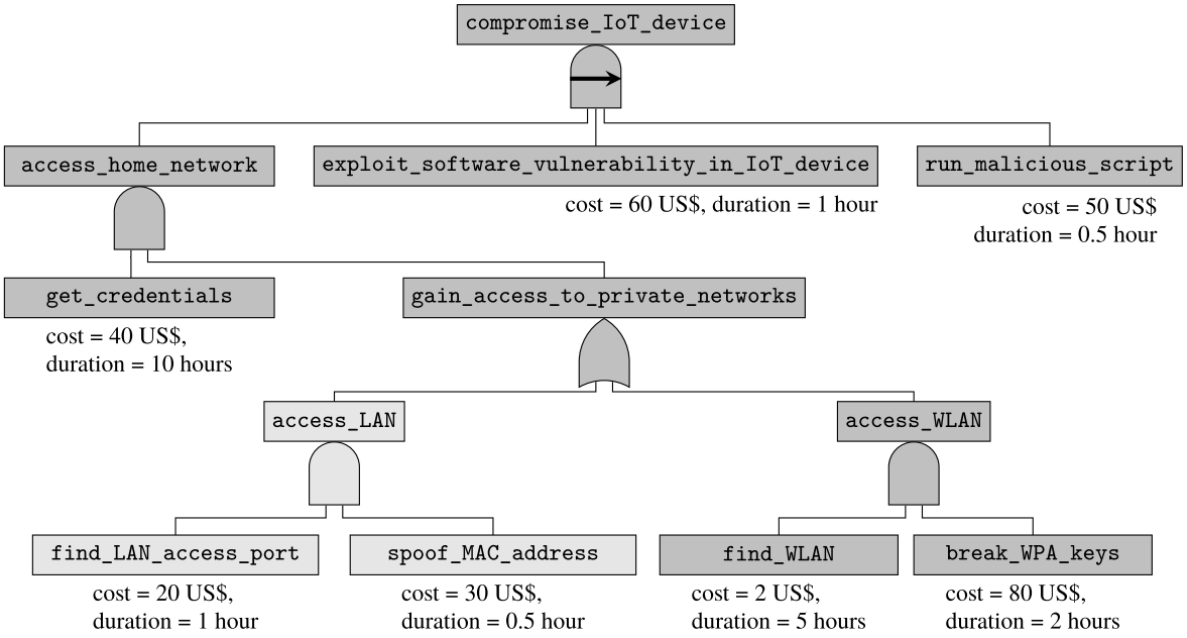



Figure 3.1: Example of an Attack Tree, taken from [Kumar et al., 2018]

In Figure 3.1, an example of an attack tree is given. The root goal of this attack is to compromise an IoT device. According to the SAND (sequential AND) gate of the root goal, there are three attack steps that need to be performed in order from left to right. First, the home network has to be accessed, then a software vulnerability has to be exploited in the IoT device and at last, a malicious script has to be run. The last two steps are basic attack steps and have a cost and duration specified for each of them. The first step however consists of two sub attack steps. The AND-gate that is used requires that both sub attacks need to be successfully performed in order for the gate task to be successful as well. The attacker needs to get the credentials of the IoT device and also needs to gain access to the private networks. Again, the first mentioned attack step is a basic attack step, but the latter again consists of two sub attacks. The gate that belongs to the latter attack is called an OR-gate and requires only either one of the sub attacks has to be successful. The attacker thus can either access the LAN or access the WLAN. Both the access LAN and access WLAN attacks are AND-gates and they both consist of two basic attack steps.

Besides the gates mentioned in the example above, attack trees can also include other gate types. A full list of gate types is given in table 4.1.

### ATTop

[Kumar et al., 2018] created ATTop[1] which is a tool to convert attack trees, fault trees or a combination of both (AFTs) from one notation to another. An attack tree created in, for instance, ADTool [Gadyatskaya et al., 2016] can be easily converted to a fault tree for DFTCalc [Arnold et al., 2013] using an ADTool file as input and selecting an ATCalc file (used by DFTCalc) as output.

[1] https://github.com/utwente-fmt/attop

Besides converting attack trees from one notation to another, ATTop can also convert the model into a UPPAAL model. ATTop creates a file that can be manually opened by UPPAAL, or call UPPAAL automatically. In the latter case, the user can specify a query that should be checked by UPPAAL and hit 'transform'. Behind the scenes, ATTop will create a UPPAAL model file as well as a UPPAAL query file and give both files to the UPPAAL verifier which will in turn try to verify the query. The result trace will be stored in the given output file and will also be shown in a message popup.

ATTop is capable of transforming one model to another fairly easily because it is based on Model Driven Engineering (MDE). In MDE, models are conforming to metamodels. In ATTop, every supported notation of attack trees, fault trees and AFTs is represented with a single unified metamodel called the attack tree metamodel (ATMM). ATMM is a generic metamodel that can be used for most types of attack trees, fault trees and AFTs based on directed acyclic graphs (DAGs). An overview of such attack trees, fault trees and AFTs is given in [Kordy et al., 2014].

### UPPAAL METAMODELS

The ATTOP tool described in the previous paragraph makes use of the UPPAAL metamodels library created by [Schivo et al., 2017]. Originally, the metamodels library was created by [Gerking, 2013] for the MechatronicUML[2] project, but they were modified so they can be used for other projects too. [Schivo et al., 2017] packaged the metamodels and additional tooling into the *EMF-based tooling for the UPPAAL model checker*[3].

The package contains metamodels to create UPPAAL templates (UTA metamodel), UPPAAL queries (UQU metamodel) and UPPAAL traces (UTR metamodel). Besides the metamodels, the package also comes with a serializer that can generate UPPAAL source files from UTA based models. The UTR metamodel comes with an additional parser that can parse actual UPPAAL generated traces and convert them into UTR based models.

**UTA metamodel** Figure 3.2 shows the UTA metamodel for UPPAAL Networks of Timed Automata (NTA). An NTA consists of one or more *Templates* and has a list of declarations assigned to it. A system declaration instantiates instances of templates and a global declaration consists of variables that can be used throughout the whole NTA. A *Template* consists of *Locations* and *Edges* and defines a timed automaton (TA). *Locations* in a TA are connected through directed *Egdes*, therefore, both incoming and outgoing edges can be assigned to locations. To identify a location, it can be assigned a *name*. Both *Locations* and *Edges* can be enriched with *Expressions*. An expression can be used as invariant for the locations and as update-statements or guards for the edges. For example, the expression $y := 0$ sets the value of $y$ to 0 when the edge is taken. An example of an invariant that can be assigned to a location is $y < 5$, which means that $y$ must be smaller than 5 in order for the automaton to be in that location. The example in Figure 2.12 shows the example of an update-expression on the edge from location *Off* to location *low*. The example does not show an example of an invariant that is assigned to a location.

**UQU metamodel** The UQU metamodel, as shown in Figure 3.3, is used to create models for UPPAAL queries. A UPPAAL query model contains a *PropertyRepository* which is a list

[2] http://www.mechatronicuml.org
[3] https://github.com/uppaal-emf/uppaal

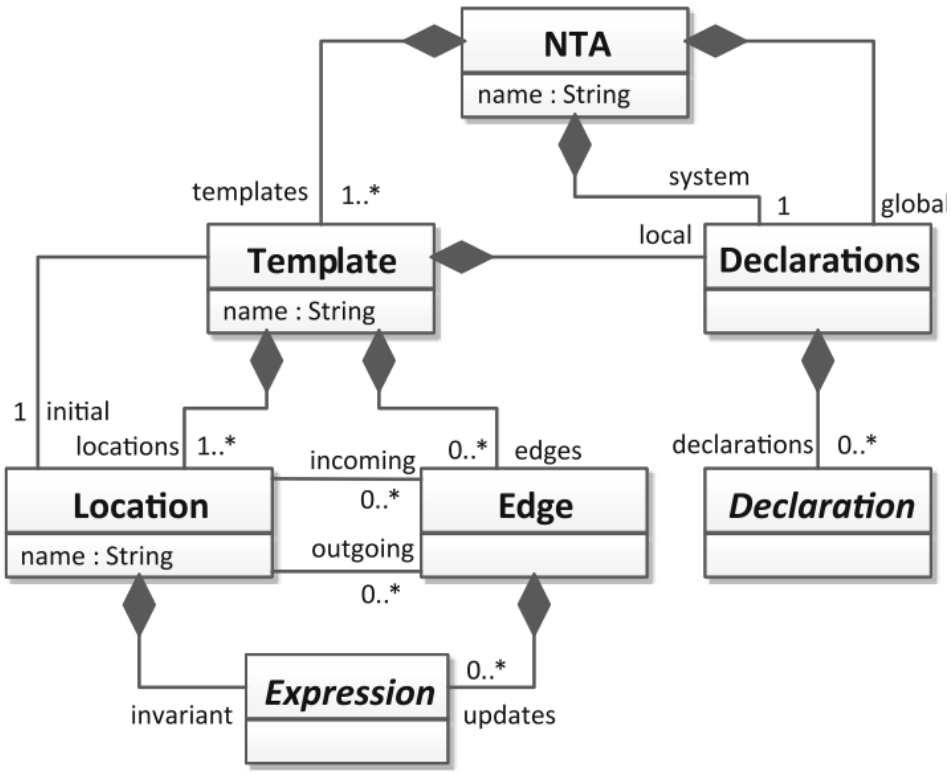



Figure 3.2: The UPPAAL Timed Automata metamodel

of *Properties* that should be verified by UPPAAL. A *Property* can be a *UnaryProperty* or a *LeadsToProperty*. A *UnaryProperty* is based on the TCTL notation. For example, **E<>** $\varphi$ and **A[]** $\varphi$ are examples of TCTL based properties. TCTL properties that can be verified with UPPAAL are further explained in Section 2.3.2.

The *Expression* type in the UQU metamodel is the same as the *Expression* type in the UTA metamodel and is therefore reused. The expression represents the property to be checked by the model checker. For instance, in **E<>** $x < 5$, $x < 5$ is the expression and it is checked if there exists a state in which the expression holds.

**UTR metamodel** When a property is verified using UPPAAL, its output can be either that the property was satisfied or not satisfied. UPPAAL can also produce a trace that shows a path that leads to the property being satisfied or not. In case of a *reachability* query, when a state is reached and the property is satisfied, a trace that leads to that state is returned, i.e. it returns the evidence. In case a *liveness* property is checked (i.e. a state should always be true), UPPAAL will return a trace in case the property was not satisfied as a counter-example.

As can be seen in Figure 3.4, a *Trace* exists of *States* and *Transitions*. Every *State* contains a list of *Locations* that are active in that state (one location per template instance). Also, a state contains the actual *clock* and *variable* values. Between *States* are *Transitions*. A *Transition* describes the transition from one state to another and can be of type *DelayTransition* or *EdgeTransition*. A *DelayTransition* describes a 'passing' of time, no edge was taken, only time has passed. An *EdgeTransition* describes that, in a template instance, the active location has changed, i.e. an edge was taken.

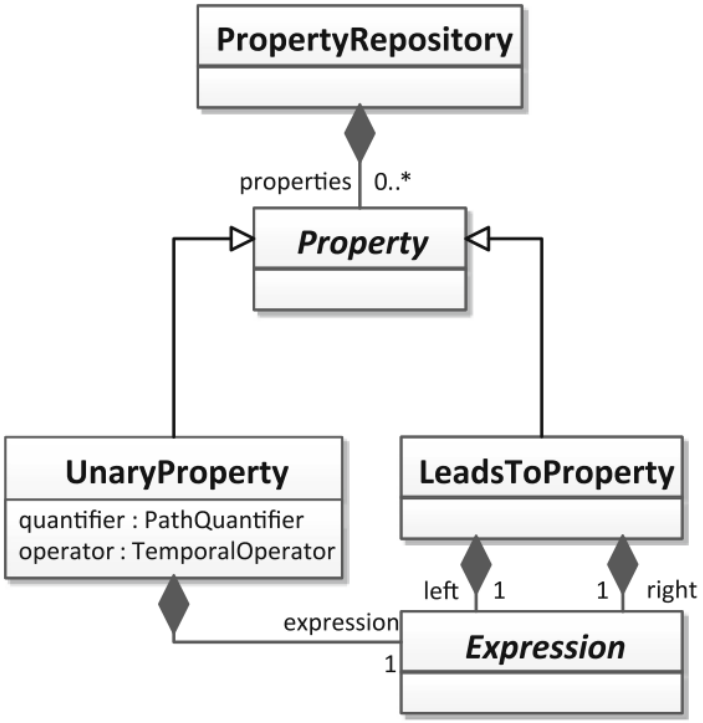



Figure 3.3: The UPPAAL Query metamodel

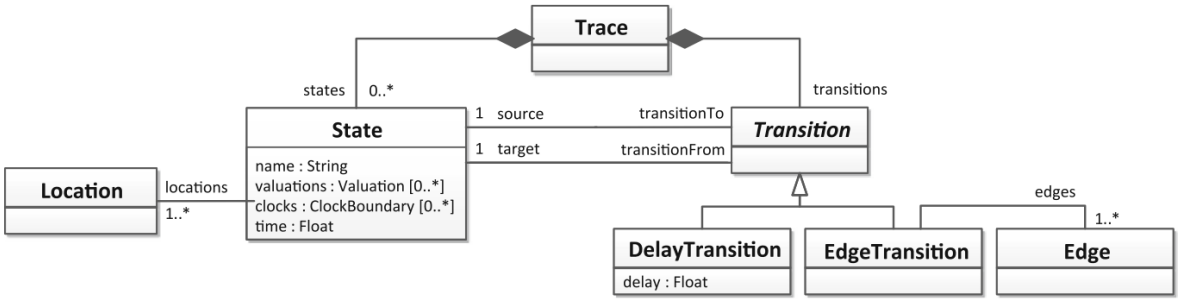



Figure 3.4: The UPPAAL Trace metamodel

# 4

# METHOD

In this chapter, we will describe the methods that were used to find answers to the research questions (see Section 1.2). For the first sub-question, *How do Task Models differ from Attack Trees?*, we will make a comparison between Attack Trees and Task models (see Section 4.1). We will compare the two by their metamodels. For Attack Trees, the metamodel is available, but for Task Models, we use the one defined by [Paternò et al., 2014].

This brings us to the next sub-question; *How can Task Models be converted to UPPAAL models using MDE?*. To answer this question, we first need to define the Task Model metamodel we are going to use (see Section 4.2). This metamodel will be a customized version of the metamodel described by [Paternò et al., 2014] because we intend to use the PCTT and BCTT notation, which are special versions of the CTT notation (see Section 2.2.3).

When we have our custom P/BCTT metamodel, we can start with defining UPPAAL representations for the elements in the Task Model metamodel (see Section 4.3). For every element (tasks and operators), we need to define a timed automaton in UPPAAL. By first creating the automata in UPPAAL, we can use UPPAAL to validate the created automata as well, which aids us in the validation later.

The next step is to transform the task models, defined in the P/BCTT metamodel, to the corresponding UPPAAL models (see Section 4.4). We will need to define transformations between elements in the task model metamodel and elements in the UPPAAL metamodel. The ETL transformation language will be used to define the transformations.

From this point, thanks to the Epsilon framework, we have a system that can read task models (in the PCTT format) and transform them to UPPAAL models. To perform model checking on the UPPAAL model, we still need to create queries in UPPAAL. This brings us to the third and last sub-question: *How can property queries be defined in a user friendly way so users do not have to write actual UPPAAL queries?*. The next step will therefore be to create a Query Generator Tool (see Section 4.5). The QGT allows users of the tool to create queries in a simple and understandable way, which abstracts away the UPPAAL queries from the user.

## 4.1. ATTACK TREES VERSUS TASK MODELS

The first question we asked ourselves in this thesis is 'How do Task Models differ from Attack Trees?'. In this section, we will make a comparison between Task Models and Attack Trees. We will do that based on the metamodel for both the Task Model and Attack Tree, so

| Gate | Meaning |
|---|---|
| AND | Requires all sub attacks to be successful, sub attacks can be started parallel |
| OR | Requires at least one sub attack to be successful, sub attacks can be started parallel |
| SAND | Same as AND, but second sub attack may only be started if the first succeeded |
| SOR | Same as OR, but second sub attack may only be performed if the first failed |
| XOR | Requires exactly one sub attack to be successful |
| TAND | Same as AND, but requires time to pass between the sub attacks |
| PAND | Same as AND, but second sub attack can only succeed after the first succeeded |
| KofN | Succeeds if K out of N sub tasks have succeeded |
| Weighted | Same as KofN, but uses weights for sub attacks |

Table 4.1: Possible gates for attack trees

it becomes easier to compare the general structure of the two types of models. Besides that, we will also take a look at the meaning of both types of models. I.e., what is a Task Model used for in comparison to an Attack Tree?

Not only will we look at the differences, but also describe the similarities among them. This will become useful when we want to reuse parts of an already created tool that converts Attack Trees into UPPAAL models, namely ATTop [Kumar et al., 2018]. The methods that they used to convert attack trees into UPPAAL models are similar to the methods that we use. Because attack trees seem to have commonalities with task models, it might be useful study the similarities. Based on the results, we might be able to reuse parts of the ATTop tool.

**Attack Tree metamodels** The metamodel for Attack Trees is given by [Kumar et al., 2018] and can be seen in Figure 4.1 and Figure 4.2. The main item in the metamodel is the *AttackTree* and acts as the placeholder from which all items can be reached. An *AttackTree* consists of a number of *Node* items. A *Node* represents an attack-step in the Attack Tree. It has a number of properties like a 'label', 'role', 'nature' and 'id'. Besides the properties, *Nodes* can also have child-*Nodes*. Child-*Nodes* and parent-*Nodes* are connected through *Edges*. This relation describes attack-steps and sub-attack-steps.

How a *Node* reacts to the state of its child-*Nodes* is determined by the type of *Connector* that is attached to the *Node*. Every *Node*, except leaf-*Nodes* must have a *Connector*. This latter rule, however, cannot be read from the metamodel. A *Connector* is often called a 'gate' and are displayed using slightly modified ANSI symbols for digital logic gates. An explanation of the gates can be found in Table 4.1.

Where non-leaf-*Nodes* can have gates, leaf-*Nodes* can have *Attributes* (see Figure 4.2). An *Attribute* has exactly one *Value* and belongs to exactly one *Domain*. A *Domain* specifies the *Type* of the attribute (i.e. a real type) and the *Purpose* of the attribute (i.e. time, cost).

**Task Model metamodels** In Figure 2.15, the metamodel for CTT models is shown [Paternò et al., 2014]. The main element in the metamodel is the *Task*. A *task* has an identifier and a name and always belongs to exactly one *TaskModel*. Additionally, a *task* may be en-

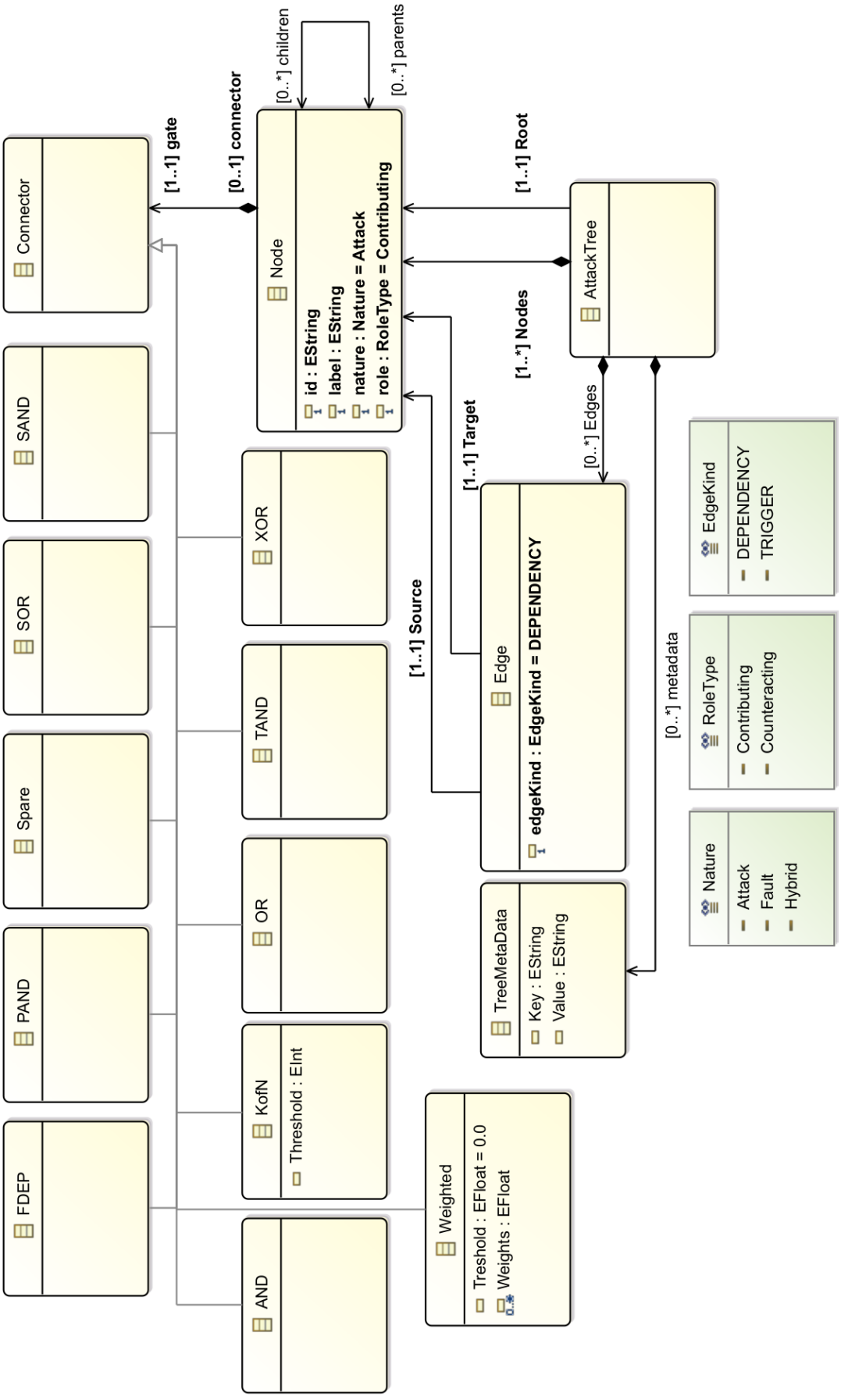



Figure 4.1: Attack Tree metamodel - Structure

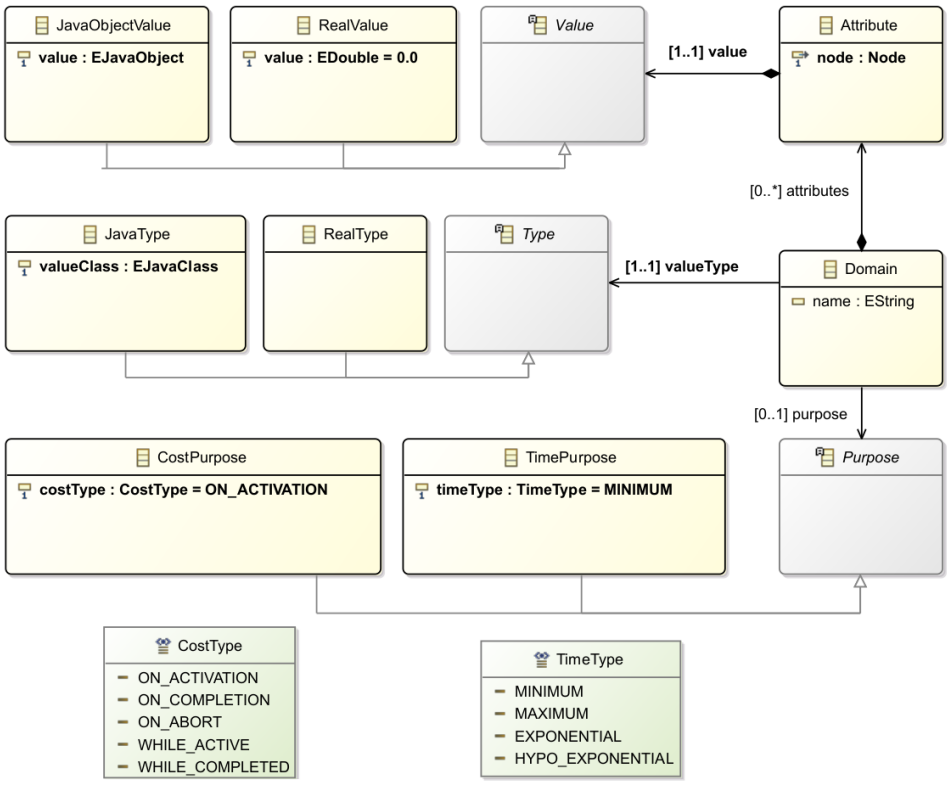



Figure 4.2: Attack Tree metamodel - Values

riched with a simple *description*. A *Task* can have references to other *Tasks*, which are called subtasks. Sibling tasks are tasks that share the same parent *task* and thus are its subtasks.

Sibling tasks are connected through *operators*. An *operator* defines when and in what order tasks are enabled for execution. There are nine types of operators, a summary of those operators is seen in Table 4.2 and a full explanation is found in Section 2.2.2.

When a *task* can be executed depends on the value of the *precondition*. A *task* can have a relation to *objects* which can be used in *preconditions*. When a *precondition* evaluates to *true*, the *Task* can be enabled for execution. *Objects* store the information that can be evaluated through *preconditions* and can be shared among *tasks*. Information in *objects* can be altered by using *postconditions*. A *postcondition* of one *task* can thus update the *precondition* of another *task*.

Furthermore, *tasks* can be marked *optional* and/or *iterative*. An *optional* tasks makes the task, as the name suggests, optional. This means that a user can **choose** whether or not to execute the task in order to fulfill the parent task. An *iterative* task is a task that can be executed an infinite number of times.

**Comparison** From the two previous paragraphs, we can see that there indeed is some overlap between Attack Trees and Task Models. Both models have a tree-like structure and start with one root node called the 'goal'. Children of the root node represent tasks or attacks, but cannot be executed by themselves. Instead, they have an attached operator which determines the behaviour of its children, i.e. which can be executed and in what order. Nodes that have **no** children are the nodes that **can** be executed. Nodes without children are called leafs. In Attack Trees, they are the attacks that can be executed by an

| TempOp | Meaning |
|---|---|
| Choice | Requires only one of the tasks to be executed. |
| OrderIndependence | Requires both tasks to be executed, but not at the same time. The order is independent. |
| Interleaving | Requires both tasks to be executed, may even be executed at the same time. |
| Synchronization | Same as *Interleaving*, but allows information to be shared among the tasks. |
| Parallel | Requires tasks to be executed at the same time. |
| Disabling | Rightmost task disables the execution of the other tasks. |
| Suspend/Resume | Rightmost task interrupts the execution of the other tasks. Other tasks resume after rightmost task is finished. |
| SequentialEnabling | Requires the execution of tasks in order from left to right, next one starting after previous is finished. |
| SequentialEnablingInfo | Same as *SequentialEnabling* but allows information to be passed to the next task(s). |

Table 4.2: Possible temporal operators for CTT task models

attacker and in Task Models, they are the tasks that can be executed by a user or system.

A big difference between Attack Trees and task models is that the *gate* in an Attack Tree belongs to the parent, whilst in Task Models, *operators* belong to the children and describe the relation between siblings. In Attack Trees, the *gate* is therefore always the same for all children. In Task Models, the *operators* between children can be different per child task.

In Attack Trees, leafs can be enriched with values of different kinds. Values can have a cost purpose or a time purpose that define what an attack costs and how long an attack takes. The leafs of Task Models can only be assigned with values that have a time purpose, i.e. the minimum and/or maximum time a task takes.

In addition to the attacks in Attack Trees, tasks in Task Models can be optional or iterative. Optional means that a user can choose whether or not to execute that task. Attack Trees do not allow attacks to be optional, an attack is something that has to be done in order for its parent attack to be successful. There is the possibility to choose between attacks (through the OR gate), but this implies there is an alternative attack. With the optional tasks in Task Models, it is not required to have an alternative task and this way, tasks can be purely optional.

The marking of iterative tasks is also something that is not available for attacks. It is not required to repeat an attack multiple times in order for its parent attack to be successful. Repetitive attacks are modelled as such that they occur multiple times in the tree. With task models, iterative tasks are useful. In case of, for instance, a user interface, it can make sense to execute a task multiple times. Take for example the task model in Figure 2.1. The task *AddNewUser* could be a perfect candidate for being marked as **iterative** as a user might be wanting to add multiple new users.

**Purpose** Besides the physical differences between Attack Trees and Task Models, there are also the differences in purpose for both models. An Attack Tree is used to model attacks [Mauw and Oostdijk, 2006; Schneier, 1999], with the intention to support security prac-

titioners by modeling the different ways in which a system can be attacked. This model can be used for documentation purposes or risk analysis for customers of a certain system. Using an Attack Tree, manufacturers of a system can determine what aspects of a system require extra protection to minimize the risk of being attacked. Because attack trees may include time and cost aspects too, one can determine which risks should have a higher priority to be eliminated.

Task Models are used for a completely different purpose. Besides for documentation (a purpose that is actually shared with Attack Trees), task models are used to model behaviours of user interfaces [Card et al., 1983; Paternò et al., 1997]. The main task, the root goal of the model, describes more like a feature than an actual goal. For instance, the root goal of the example in Figure 2.1 could be *Manage users*. Manage users is not something that will eventually be 'done', but more something that describes the purpose of the UI. The purpose of a task model is therefore also not describing some sort of scenario that leads to the completion of the root task (main goal). Instead, it describes the task of the UI, which is in the case of the example to *manage users*.

## 4.2. Definition of the custom CTT metamodel

**CTT, PCTT or BCTT** For our tool, we are going to make use of the Binary CTT notation. The BCTT notation is semantically the same as the CTT notation, but is easier to handle because every task has exactly zero or two subtasks. This ensures that we do not have to handle priorities later when transforming the model to a UPPAAL model. The tool with which we create CTT task models, CTTE, is not capable of generating BCTT models directly. It can, however, convert CTT task models to Prioritized CTT task models to get rid of priority issues. The conversion from PCTT to BCTT models can then be done by our tool.

The metamodel for both the PCTT and BCTT task model notations could be the same. In both PCTT and BCTT task model notations, the TempOp belongs to the parent task instead of the subtasks. The metamodel is therefore the same, except for the multiplicity (cardinality) values between tasks and subtask; in a BCTT, a parent task has exactly two subtasks whilst in a PCTT, a parent task can have two or more subtasks. Because of them being practically the same, we will define a metamodel for the PCTT model, which can then also be used for BCTT models. In fact, the first step in our conversion tool will be to convert the PCTT from the CTTE tool to a BCTT model.

**Simplified CTT** In Figure 2.15, the metamodel for CTT is given. Besides the fact that the metamodel allows for multiple TempOps among subtasks, it also describes the usage of *(Domain) Objects*. *Objects* in CTT can be used in pre/post-conditions for tasks. How this feature is supposed to be used is not very well documented. This, in turn, can lead to a non-consistent usage of this feature. CTTE, the tool with which one can create CTT models, does not validate *Objects* either. An object that is referenced by two Tasks for instance can be misspelled in one of the Tasks which can result in unexpected behaviours of the model. As a result, we have decided not to feature *Objects* in our tool.

**New PCTT metamodels** In the previous paragraphs we have mentioned that the metamodel for PCTT can be based on the metamodel for the regular CTT, but that edges between elements need to be re-assigned. Also, we need to skip some elements that we decided not to use for now.

When we look at the definition for the PCTT notation (Definition 4), we can see that a *Task* is assigned zero or more subtasks, a TempOp is assigned to every non-leaf task and Time Performance (TP) is assigned to every leaf task. Based on this definition, we designed the metamodel in Figure 4.3.

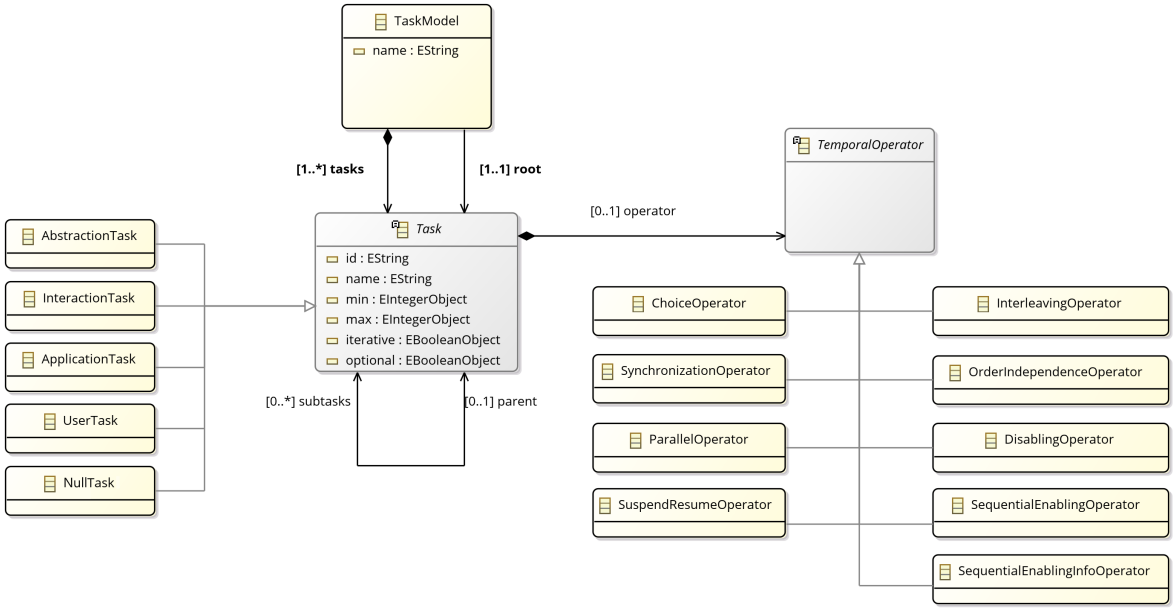



Figure 4.3: ConcurTaskTree metamodels

In the metamodel of Figure 4.3, we moved the *TempOp* from being an associative element of the relation between *Task* and its sibling to being related directly to the *Task* element as a composition (a *task*, when non-leaf, must have a *TempOp* assigned to it).

Furthermore, we state that a *TaskModel* contains a collection of *Task* elements and that **one** of those *Task* elements acts as the **root** *Task*. In the original CTT notation, the root task was not described and could therefore only be found by checking for the one task element that had no parent task.

For the PCTT metamodel, we aligned the types of *TempOps* with the ones that are available in the CTTE tool. This means that we added the *Parallel* operator and renamed the *Concurrency* operator to *Interleaving*. Additionally, we removed the *priority* attribute from the *Operator* element as priority is already taken care of in a PCTT.

As for the *Task* element in the metamodel, we removed and renamed some attributes. The *Frequency* attribute is replaced by the *min* and *max* attributes and the pre/post-conditions are skipped. The attributes *Platform* and *Description* are not relevant for model-checking and are therefore also skipped. They are useful when the model is used to generate user interface, but has no impact on the execution of such a model. The *name* and *id* however, identify the task and are used in the model checking for traceability purposes.

Based on the comparison with Attack Trees (see Section 4.1), it might seem obvious to also create a *Values* part for the PCTT metamodel. The additional metamodel could then take care of the values in the task model. Because we only have two values in the task model; min- and max time, adding an entire metamodel would make things over complicated. We therefore decided to encapsulate those values in the PCTT metamodel. Eventually, this will also simplify the model transformations.

## 4.3. DEFINE TASK MODEL ELEMENTS IN UPPAAL

In this section, we are going to describe the PCTT task model element representations as UPPAAL templates. Both the UPPAAL templates and UPPAAL systems that configure the templates are eventually going to be generated by our tool. We will describe the most important templates in detail in this Section. Other templates are written in a similar manner. For detailed versions of the templates, the source code is available in Github[1].

We have defined a UPPAAL template for every task type in the PCTT metamodel. This means that we have created templates for the simple leaf task, but also for the more complex non-leaf tasks that are defined by the TempOps that are assigned to them. Every template can be instantiated with parameters that configures the template. For instance, a leaf task template is instantiated with an *id*, a boolean that marks the task *iterative*, an optional *min_time* value and an optional *max_time* value. The parameters for non-leaf task templates are different. The first two parameters are the same, i.e. they are also given an *id* and a boolean that marks the task *iterative*. Further, non-leaf task templates are given the *id* of its left subtask and the *id* of its right subtask.

The *TopLevel* template (see Figure 4.4) is special and is used to kick-start the task model. The *TopLevel* template can only be instantiated once and is given the id of the *root* task of the task model as a parameter. Besides kick-starting the task model, it also keeps track of the total time and actual running time. When the top level task is done, both the total and running time clocks are stopped so they can be queried.

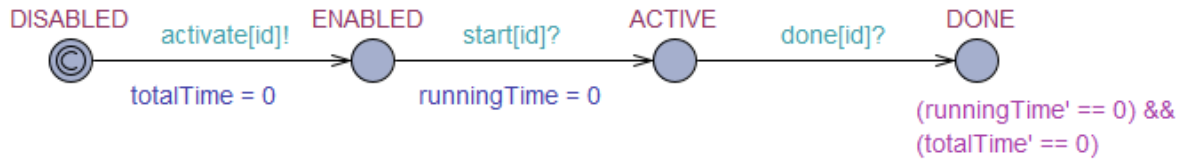



Figure 4.4: Implementation of the top level task

Using the parameters of templates, we can specify the configuration of template instances in UPPAAL. In the example of Listing 1, the system declaration of the task model in 2.5 is given. The example shows the instantiation of different kinds of templates. First, the *TopLevel* template is instantiated with the id of the *root* task as its parameter. Then, for every task in the model, the corresponding Template is instantiated. The task model contains three *Interaction Leaf Tasks*; *task_1* (id=1), *task_2* (id=4) and *task_3* (id=5), one *Application Leaf Task*; *task_4* (id=6), contains two *Interleaving* tasks; *task_0* (id=0, left=1, right=2) and *tb* (id=2, left=3, right=6) and contains one *Choice* task; *ta* (id=3, left=4, right=5). None of the tasks is marked as *iterative* as all tasks are instantiated with **false** for the second parameter. The instances are assigned to the system by using the **system** keyword (see last line of Listing 1.)

### 4.3.1. LEAF TASKS

Tasks that have no subtasks are called leaf tasks. Such a leaf task is also defined as a task that has no *TempOp* assigned to it. Leaf tasks are the most basic tasks and can be automatically started in case of an *Application Task* or manually started in case of an *Interaction task*

[1] https://github.com/egbertpostma/ctt-temporal-operators

```
top_level = TopLevel(0);
task_0    = Interleaving(0, false, 1, 2);
task_1    = InteractionTask(1, false, -1, -1);
tb        = Interleaving(2, false, 3, 6);
ta        = Choice(3, false, 4, 5);
task_2    = InteractionTask(4, false, -1, -1);
task_3    = InteractionTask(5, false, -1, -1);
task_4    = ApplicationTask(6, false, -1, -1);

system top_level, task_0, task_1, tb, ta, task_2, task_3, task_4;
```

Listing 1: UPPAAL System declaration for task model in Figure 2.5

or *User task*. (Note that leaf tasks cannot be of type *Abstraction task* by design). An *Application task* is automatically started when it becomes enabled, i.e. there is no time between enabling and starting the task. An *Interaction task* or *User task* is not automatically started when it becomes enabled, i.e. it can take time before the task is started after it is enabled (thus also indefinitely).

We have shown the state model for leaf tasks in CTT in Figure 2.8. The figure shows the states in which a leaf task can be, as well as the events between the states. What is missing in the figure are the time specific elements and the disabling transitions that go from every state to the 'Disabled' state.

### INTERACTION/USER TASK

Figure 4.5 shows the interaction leaf task representation as a UPPAAL template. The model of the template is built upon the state model in Figure 2.8. The model has two extra locations with respect to the state model. First, the *Suspended* location is split up in two independent locations which is due to the fact that when the model is resumed from the *Suspended* state, it should resume to the state it was in when it got suspended. By using two *Suspended* locations, it is guaranteed that the model is returned to the location prior to suspending. The second extra location is the 'committed' location after the *Active* location. This location is used to 'exit' the *Active* location and move to either the *Done* location when `iterative==false` or the *Enabled* location when `iterative==true`. This way, the guard that handles the *min_time* check can be specified only once instead of twice.

An interaction leaf task becomes *enabled* when it receives an activation signal (`activate[id]?`) from its parent task (or the *TopLevel* task if it is the root task). When the task is in the *Enabled* state, it can start itself or be suspended by its parent. When it starts, it broadcasts a start signal (`start[id]!`) which can be picked up by its parent. The model is now moved to the *Active* state and its *time* variable is reset to 0 (`time = 0`). An *Active* task can be suspended by its parent task (`suspend[id]?`). When the task is in the suspended location, the `time' == 0` invariant stops the clock so the *time* variable is not increased. When the task is resumed (`resume[id]?`), the clock is automatically re-enabled. The task can be in the *Active* location as long as it satisfies the invariant that is set on the location. This means that it can stay in the *Active* location indefinitely if `max_time == -1` or as long as `time <= max_time` if max_time is set to a value other than -1. Also, the *Active* location cannot be left if the guard on the exit transition is not satisfied. The *Active* location can be

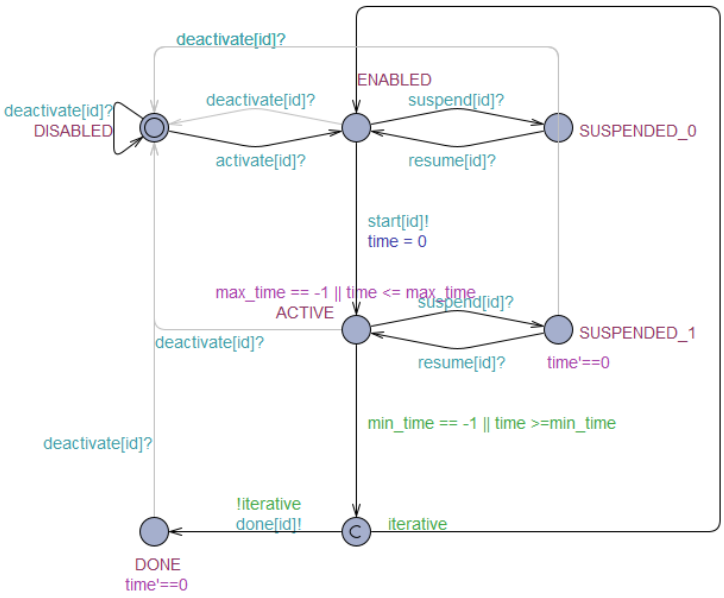



Figure 4.5: The representation of a CTT interaction leaf task in UPPAAL.

left when `min_time == -1` or `time >= min_time` if min_time is set to a value other than -1. It depends on the value of the *iterative* variable if the task is moving to the *Done* location or is re-iterated to the *Enabled* location again. If the task is in the *Done* location, the clock is stopped (`time' == 0`) as the task is no longer running anymore. Deactivating the task is possible from all states, hence from all states there is a `deactivate[id]?` that leads to the *Disabled* state.

### Application task

Figure 4.6 shows the *Application Leaf Task* representation as a UPPAAL template. The *Application Leaf Task* is largely equal to the *Interaction Leaf Task* but the major difference is that it is automatically executed when it becomes enabled. The *Enabled* location is `committed` which means that time cannot pass in that location. Because of this, there is no need to be able to suspend or disable the task from that location anymore too so the *Suspended_0* location is removed as well as the transition to the *Disabled* location.

As mentioned before, the *Application Leaf Task* is quite similar to the interaction leaf task. This means that, in general, the task follows the same state model as the interaction leaf task. Now, when the task is activated by its parent (`activate[id]?`), it immediately notifies its parent that it has become active (`start[id]!`). An *Application Leaf Task* is therefore always executed when it becomes enabled, which is not necessarily the case with interaction leaf tasks as they can remain enabled for ever and never get executed at all.

### Null task

A *Null Task* is not an actual existing task in CTT. It was introduced to replace tasks that are optional. In combination with the choice operator, a *Null Task* can be used to mimic the behaviour of an optional task as was explained in Section 2.2.2 Figure 2.2. A *Null Task* is meaningless and therefore has no parameters for time in comparison to the regular leaf tasks. The representation of the *Null Task* in UPPAAL is shown in Figure 4.7.

The state model for a *Null Task* is almost the same as the one for the regular leaf tasks. The only differences are that a *Null Task* cannot be iterative and that it cannot be active,

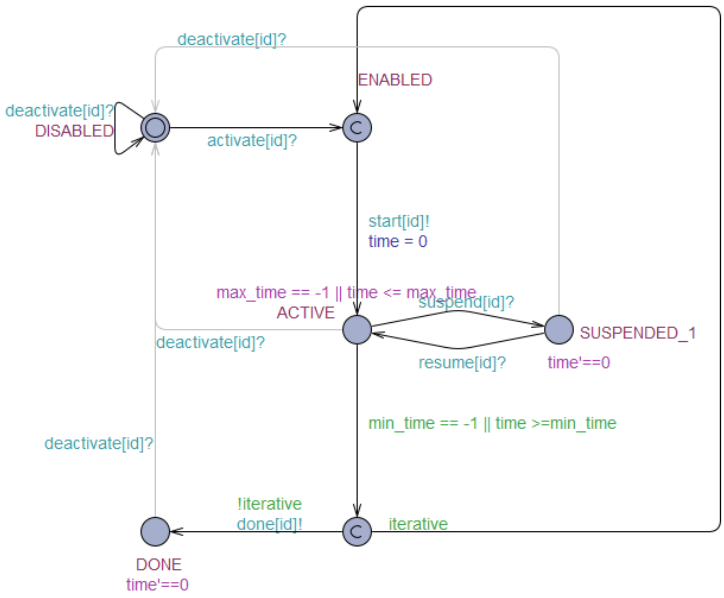



Figure 4.6: The representation of a CTT application leaf task in UPPAAL.

i.e. when it is started (`start[id]!`), it is also immediately done (`done[id]!`). A *Null Task* starts, just like regular leaf tasks, in the *Disabled* state and can be enabled by its parent (`activate[i]!`). When it is in the *Enabled* state, the task can be suspended and resumed by its parent (`suspend[id]?` and `resume[id]?`).

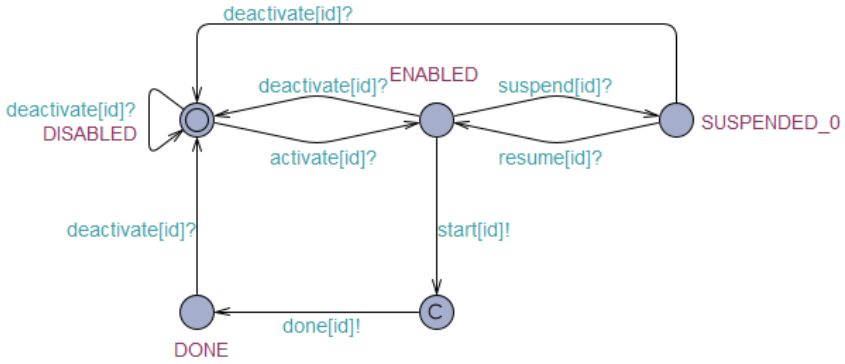



Figure 4.7: The representation of a null task in UPPAAL.

### 4.3.2. NON-LEAF TASKS

Non-leaf tasks are tasks that have subtasks. Non-leaf tasks are also tasks that have a *TempOp* assigned to them. It would, therefore, be right to say that the non-leaf task is *of type <TempOp>*, or it is *a <TempOp> task*. A non-leaf task that has a *SequentialEnabling* TempOp assigned to it can thus also be called a *SequentialEnabling task*.

#### SEQUENTIALENABLING TASK

The template of Figure 4.8 shows the UPPAAl representation of a *SequentialEnabling task*. A *SequentialEnabling task* is a task that, after it is enabled, enables its left subtask. When its left subtask is done, it enables its right subtask. When the right subtask is also done, the

*SequentialEnabling task* is done. The state model for the *SequentialEnabling task* broadly follows the state model for leaf tasks. The difference is that it cannot start itself, as well as it is in charge of its subtasks.

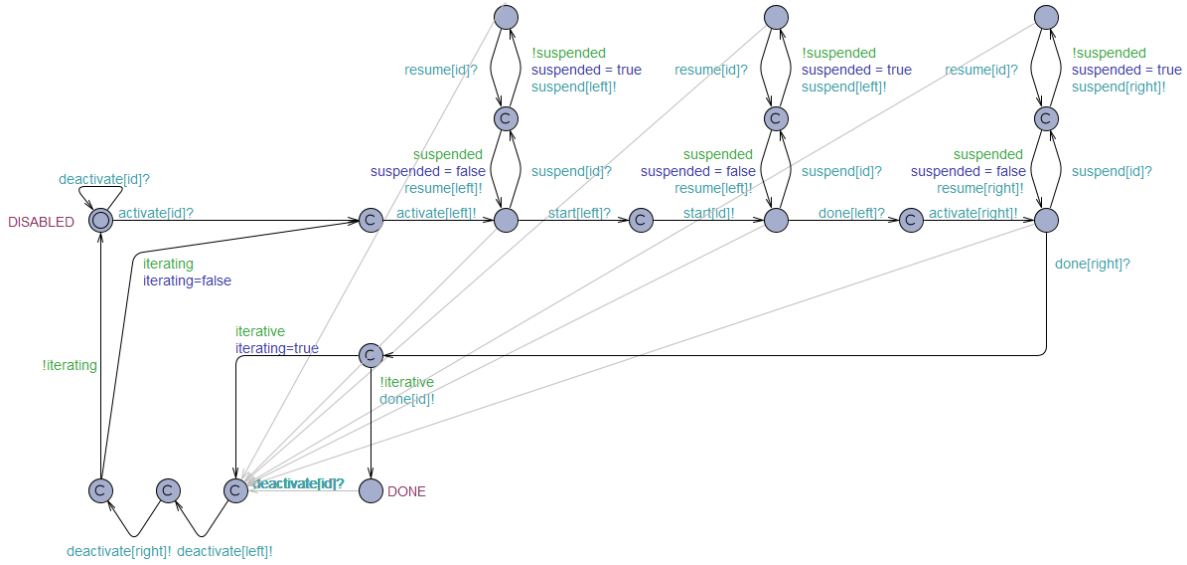



Figure 4.8: The representation of a CTT SequentialEnabling task in UPPAAL.

As one can see in Figure 4.8, the sequence of states is similar to that of a leaf task, only the *Enabled* state and *Active* state are replaced by states specific to the *SequentialEnabling task*. When the task is activated by its parent (`activate[id]?`), it immediately activates its left subtask (`activate[left]!`). The task is now in the *Enabled* state and waits for its left subtask to be started. At this moment, it is also possible for the task to be suspended by its parent (`suspend[id]?`). In this case, the task will, in turn, suspend its left subtask (`suspend[left]!`). Note that the right subtask is not suspended, as it was not yet enabled. When the task is resumed (`resume[id]?`), the left subtask is also immediately resumed (`resume[left]!`). If the left subtask is started (`start[left]?`), the task will notify its parent task that it has started (`start[id]!`). The task will now wait for the left subtask to become done (`done[left]?`). Also, in this waiting state, the task can be suspended and resumed. When the left subtask is done, the task will immediately enable the right subtask (`activate[right]!`). Because the task already notified its parent that it has started, this is not repeated. For now, the task will wait until the right task is done (`done[right]?`). Also now, the task can be suspended by its parent. However, now only the right subtask will be suspended instead of the left subtask as it has already finished. When the right subtask is done, the task will check whether to iterate or to finish. If `iterative == true`, the task will reset its subtasks and reactivate its left-subtask. If `iterative == false`, the task moves to the *Done* state and notify its parent task that the task has finished (`done[id]!`).

### CHOICE TASK

In Figure 4.9, the template for the choice task representation in UPPAAL is given. A choice task is a task that, after it is enabled, enables both its subtasks. Then, when one subtask is started, the other subtask is disabled. When the started subtask becomes done, the choice task becomes done as well.

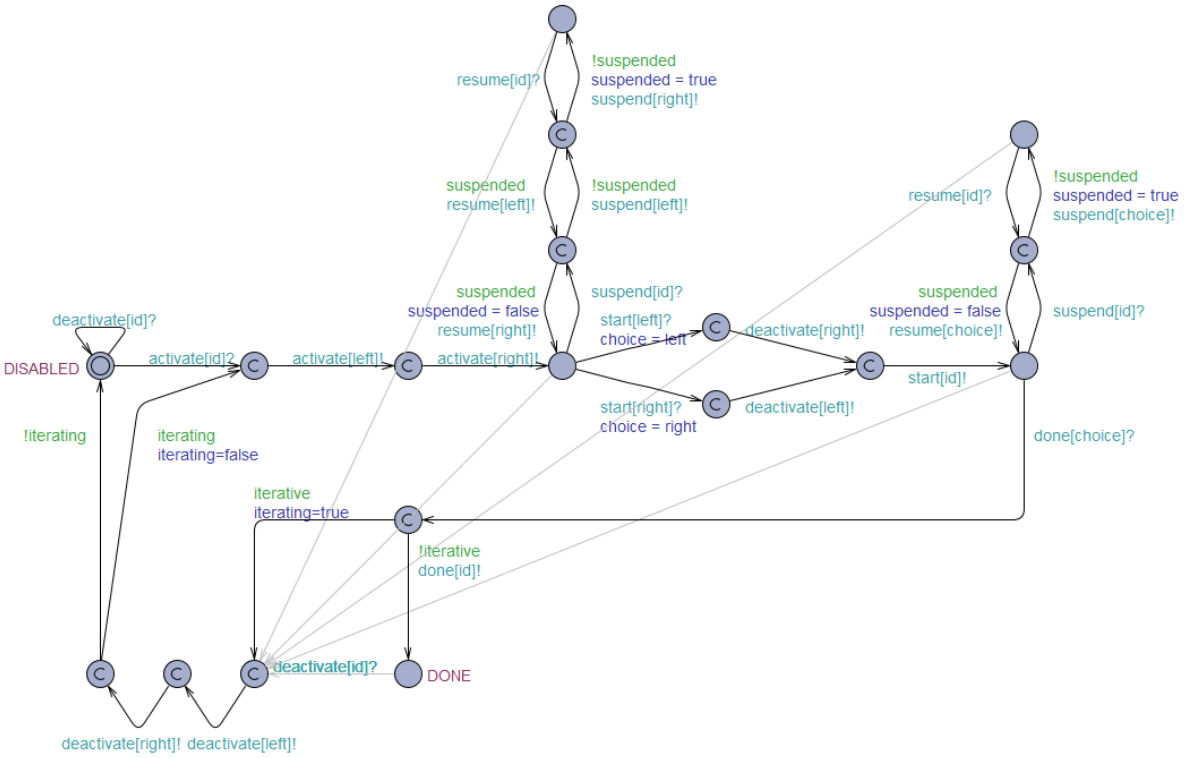



Figure 4.9: The representation of a CTT Choice task in UPPAAL.

The initial location of the template is the *Disabled* location. From there, it can be enabled by its parent (`activate[id]?`) which, in turn, activates both its subtasks (`activate[left]!` and `activate[right]!`). When one of the subtasks is started (`start[left]?` or `start[right]?`), the choice is 'cached' by setting the `choice` variable to the id of the started subtask. Immediately after that, the other subtask is disabled (`deactivate[right]!` or `deactivate[left]!` respectively) and the parent task is signalled that the choice task has started (`start[id]!`). The template is now in the *Active* state and waits for the chosen subtask to finish (`done[choice]?`). When the subtask is finished, the choice task will move to the *Done* state (`iterative == false`) and signal its parent that it is done (`done[id]!`), or iterate and begin all over again (`iterative == true`). When the task is in a waiting location (i.e. it is waiting for its subtasks to start or finish), the choice task can be suspended.

### INTERLEAVING TASK

An interleaving task is a task that, when it becomes enabled, enables both its subtasks. When one of the subtasks is started, the interleaving task notifies its parent that it has started. The interleaving task is done when both its subtasks are done. The order of the subtasks to be done does not matter. An interleaving task represents a task of which both subtasks can be active at the same time and finish in a random order. Figure 4.10 shows the UPPAAL representation of an interleaving task.

The interleaving task begins, like the other non-leaf tasks, in the *Disabled* state. When it is enabled by its parents (`activate[id]?`), both its subtasks are immediately activated (`activate[left]!` and `activate[right]!`). When one of the subtasks is started (`start[left]?` or `start[right]?`), the interleaving task notifies its parent that it has started (`start[id]!`). The task now waits for one of the subtasks to become done (`done[left]?` or `done[right]?`).

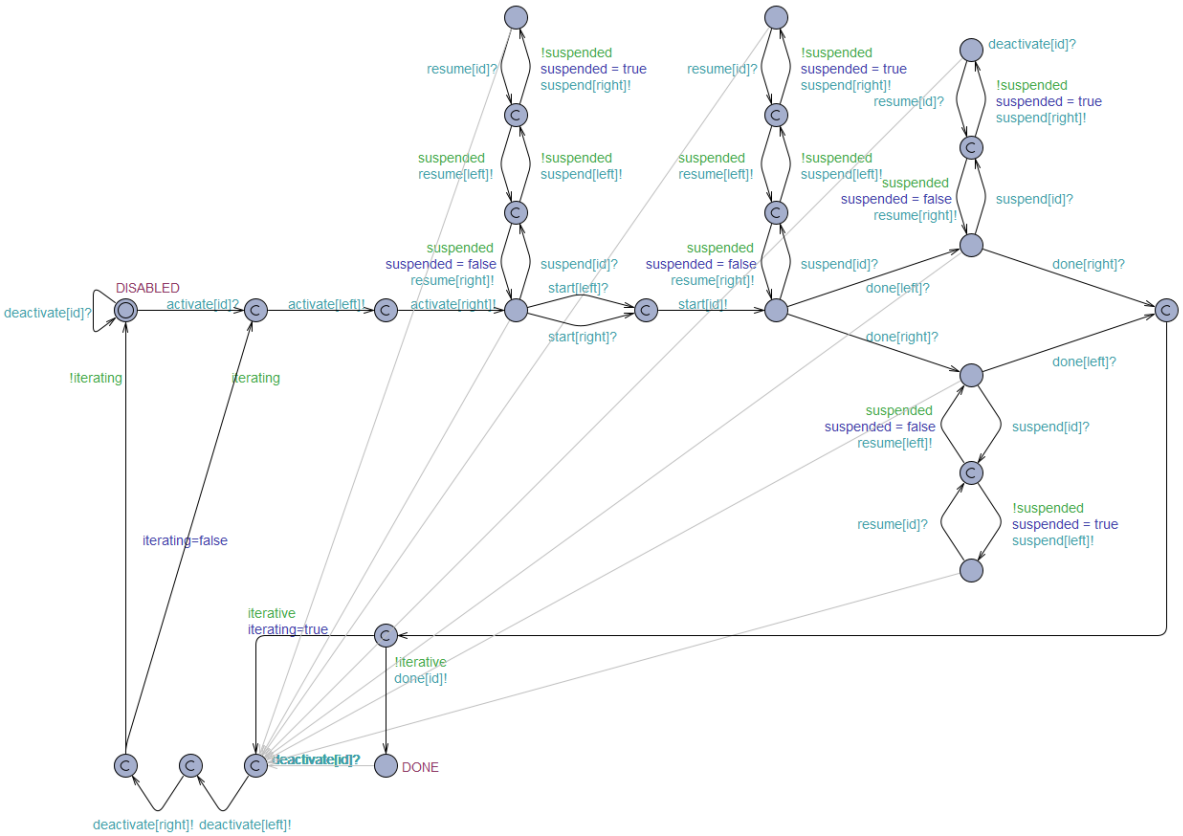



Figure 4.10: The representation of a CTT Interleaving task in UPPAAL.

If the left subtask finished first, it will then wait for the right subtask to finish. Else, if the right subtask finished first, it will wait for the left subtask to finish subsequently. When both subtasks are finished, the interleaving task either becomes done (`done[id]!`) if it was not iterative (`iterative == false`) or restarts if it was iterative (`iterative == true`). In any of the states in which the interleaving task is waiting (i.e. waiting for subtasks to start or finish), it can be suspended by its parent. Depending on the state, the interleaving task will then suspend both subtasks or, if one of them has already finished, suspend the other subtask. The same holds for resuming.

### Parallel task

A parallel task is a task that requires its subtasks to start simultaneously. After that, the order in which the tasks finish is not important which makes a parallel task similar to an interleaving task. Some editors lack the parallel operator because of the similarity to interleaving task (for instance the CTTE tool). Because of this, we are also treating parallel tasks as interleaving tasks. The UPPAAL representation for a parallel task is therefore the same as the representation for interleaving tasks. We do create a dedicated template for parallel tasks for future purposes, but for now it contains the same template as the interleaving template.

### Disabling task

The disabling task is a task that, whenever its right subtask starts, disables its left subtask. This means that a disabling task can only become done whenever its right subtasks becomes done and that the state of the left subtask is irrelevant. Often, the left subtask is an

iterative task and the disabling task is used to disable the iterative task (as it never becomes done by itself). This also means that it can also be possible that the left subtask is never ever started.

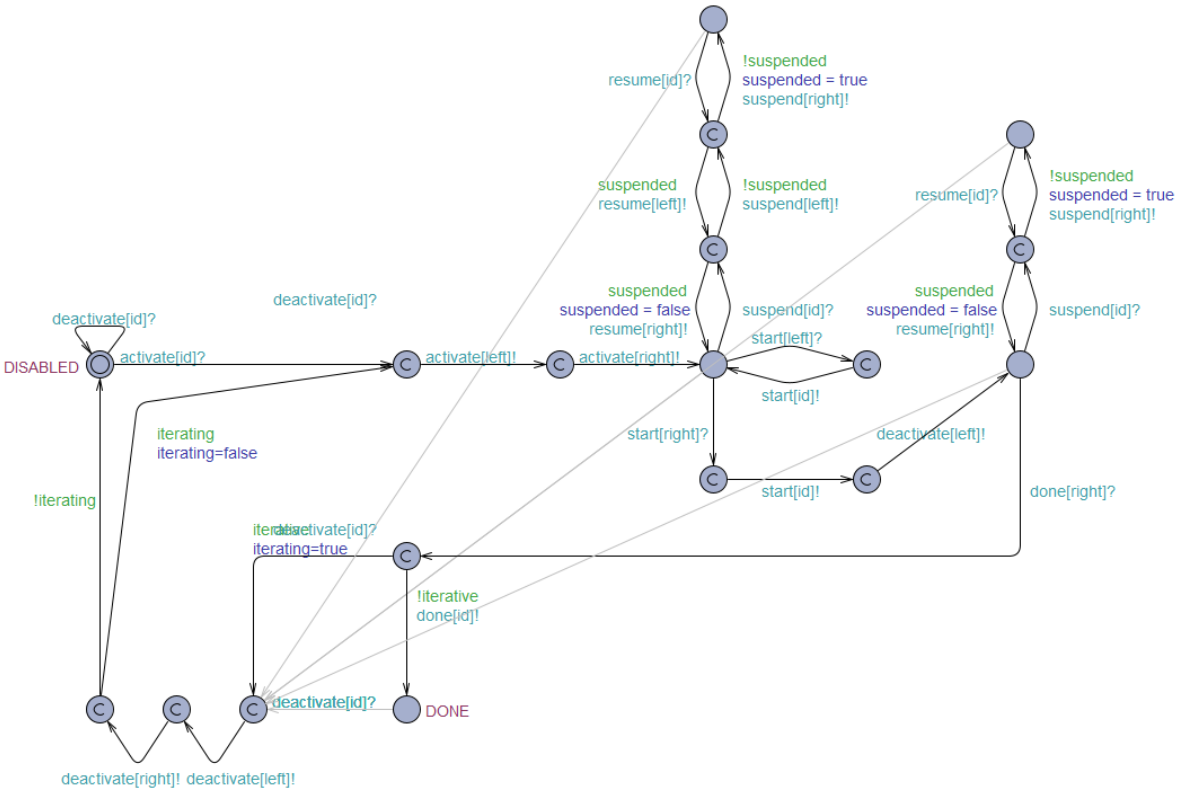



Figure 4.11: The representation of a CTT Disabling task in UPPAAL.

As can be seen in Figure 4.11, the task starts in the *Disabled* state and becomes enabled whenever its parent enables it (`activate[id]?`). Immediately, both subtasks are enabled (`activate[left]!` and `activate[right]!`). Now the disabling task wait for one of its subtasks to start (`start[left]?` or `start[right]?`) after which it notifies its parent that it has started (`start[id]!`). When the left subtask started, nothing actually changes, but when the right subtask starts, the left subtask is immediately disabled (`deactivate[left]!`). The disabling task will wait for the right subtask to become done (`done[right]?`). It depends on whether the disabling task was marked as *iterative* if it will notify its parent that it is done (`done[id]!` and `iterative == false`) or if it will iterate and re-enabled both subtasks (`iterative == true`). Whenever the disabling is waiting for actions of its subtasks, it can be suspended (and later resumed).

### SUSPEND/RESUME TASK

Probably the most complex task in UPPAAL is the suspend/resume task. The semantics of it are not complex but the number of locations and transitions in UPPAAL can make it look complex. In a suspend/resume task, the left subtask is suspended when the right subtask is started. When the right subtask is done, the left subtask is resumed. The suspend/resume task is done whenever the left subtask is done. Also, the right subtask can be executed an infinite number of times, i.e. it is iterative. The UPPAAL representation of the suspend/resume task can be seen in Figure 4.12.

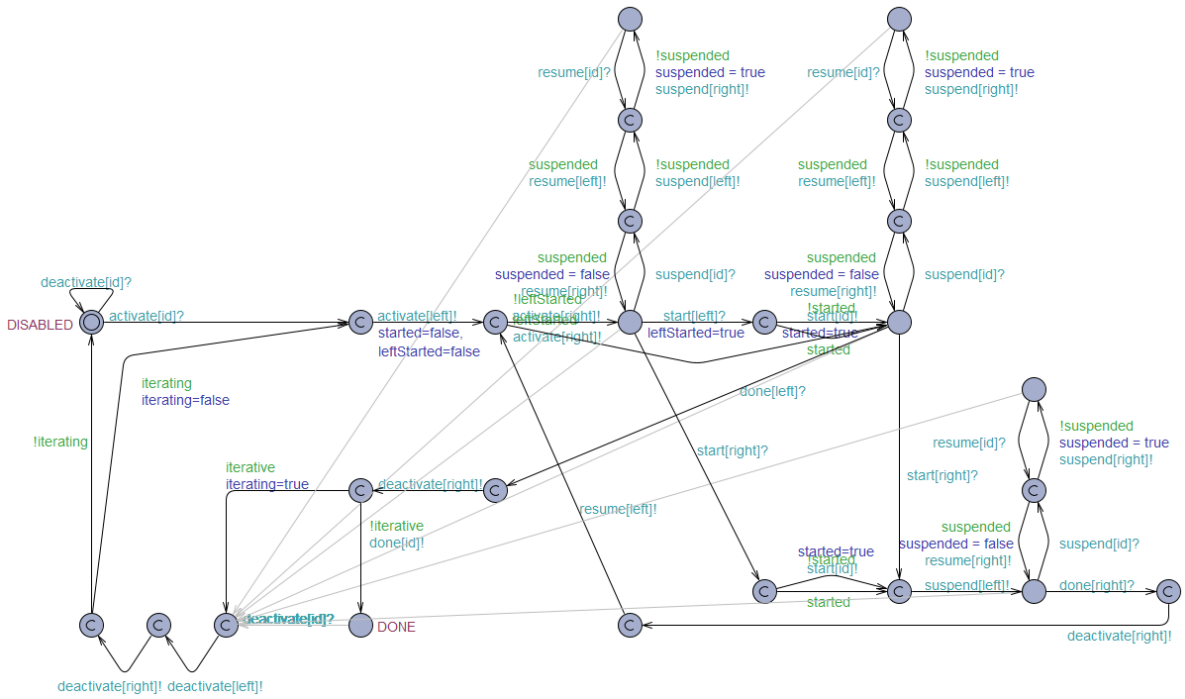



Figure 4.12: The representation of a CTT Suspend/Resume task in UPPAAL.

The suspend/resume task is enabled when its parent tells it to (`activate[id]?`). It immediately enables both its subtasks (`activate[left]!` and `activate[right]!`). If either the left or right subtasks is started (`start[left]?` or `start[right]?`), the suspend/resume task will notify its parent that it has started (`start[id]!`). When the left subtask is done (`done[left]?`), the suspend/resume task is done (`iterative == false`) or restarted (`iterative == true`). As long as the left subtask is not done (or even started), the right subtask can be started (`start[right]?`). In case the left subtask is started when the right subtask is started, the left subtask will be suspended (`suspend[left]!`). The left subtask is now temporarily inactive and cannot and will not change state. Whenever the right subtask is done (`done[right]?`), the left subtask will be resumed (`resume[left]!`). The left subtask can be suspended when its either in the *Enabled* state or in the *Active* state. This means that either the left subtask had already started when it was suspended or it was waiting to be started. When it had already started, the suspend/resume task should not again wait for the left subtask to start after it was resumed. The boolean `leftStarted` is used to keep track of this.

## 4.4. Converting PCTT to UPPAAL

The following step in creating our tool is to define the model transformations. We have the metamodel for the PCTT model and also the metamodel for UPPAAL. We know how the representation in UPPAAL is going to be like. What is left is the transformation from PCTT task models to the UPPAAL systems.

### 4.4.1. CTTE FILES TO BCTT MODELS

The first transformation is between the output files of the CTTE tool to models that conform to the PCTT metamodel. CTTE files are based on the XML file format and need to be converted to PCTT metamodel. For the model transformations we are going to use the Eclipse Modeling Framework (EMF) in combination with Epsilon. Epsilon, by default, comes with a metamodel for XML and is also capable of reading in XML files. Because we also require the CTT to be binary (BCTT), the first transformation that we need is thus from XML to BCTT. This is possible as the metamodel for BCTTs is the same as the PCTT metamodel.

```
<TaskModel NameTaskModelID="Task_0">
    <Task Identifier="Task_0"
          Category="interaction"
          Iterative="false"
          Optional="false">
        <Name>Task 0</Name>
        <SubTask>
            <Task Identifier="Task_1"
                  Category="interaction"
                  Iterative="false"
                  Optional="false">
                <Name>Task 1</Name>
                <TemporalOperator name="Parallel"/>
            </Task>
            <Task Identifier="Task_2"
                  Category="interaction"
                  Iterative="false"
                  Optional="false">
                <Name>Task 2</Name>
            </Task>
        </SubTask>
    </Task>
</TaskModel>
```

Listing 2: Example CTTE output file (XML)

For our tool, we expect CTTE to output priority task trees. A valid CTTE XML file consists of exactly one `TaskModel` element. Next, the `TaskModel` element contains exactly one `Task` which is implicitly the root task. If a `Task` contains subtasks, then the `Task` element contains a `SubTask` element which parents the subtasks as new `Task` elements. Every `Task` elements has a unique `Identifier` attribute. Also, all the `Task` elements that are children of another task share the same `TemporalOperator` element (true if the task tree is a priority task tree). An example of a CTTE XML file is seen in Listing 2. We will use Epsilon's EVL language to validate the CTTE output files (see Listing 3 for an example EVL file). This way, we know for sure that the models that we are going to transform are in the correct form.

After the XML output of CTTE is found to be valid, we can proceed with the model transformation. For this, we use Epsilon's ETL language. In ETL, one specifies transformation

```
// Part of CTTXML.evl
context CTTXML!t_Task {
    constraint SubtasksHaveSameOperator {
        guard: self.e_SubTask != null
        check {
            var operator =
             ↪ self.e_SubTask.c_Task.first().e_TemporalOperator.a_name;
            return self.e_SubTask.c_Task.atLeastNMatch(a |
             ↪ (a.e_TemporalOperator != null and
             ↪ a.e_TemporalOperator.a_name == operator),
             ↪ self.e_SubTask.c_Task.size()-1);
        }
        message: "The subtasks of Task " + self.a_Identifier + " should
         ↪ share the same temporal operator."
    }
}
```

Listing 3: Example EVL file to validate CTTE output files (XML) - this constraint validates that all subtasks of a task have the same TempOp. First, the operator is retrieved from the first subtask. Next, we check if at least all minus one subtasks also have that operator. Minus one because the last subtask may not have a temporal operator assigned. The constraint has a guard that makes sure that only tasks that actually have subtasks are checked. If the check fails, the message is outputted and the validation fails.

rules that transform elements from one metamodel to elements of the other metamodel. In case of the CTTE XML to BCTT, we need to create a rule that transforms the taskmodel element in XML to a taskmodel element in BCTT. Listing 4 shows the main transformation rule for converting CTTE XML files to BCTT models. The operation shown in Listing 5 is used to traverse the CTTE XML task tree and create BCTT task elements for every XML task.

When the CTTE XML output is converted to a BCTT model, we need to verify that the conversion was successful. For this, we are again using Epsilon's EVL language. We validate that the taskmodel element has a 'root' task assigned and that the 'root' task actually is a root task, i.e. it has no parent task. We also validate the created task elements by checking whether the id of the task is unique, all non-leaf tasks have an operator assigned to them and that all non-leaf tasks have exactly two subtasks (i.e. the tree is binary).

### 4.4.2. BCTT MODELS TO UPPAAL SYSTEMS

In Section 4.4.1 we have converted CTTE based task models into BCTT models. Those BCTT models can now be converted into UPPAAL systems. For this, we are again making use of Epsilon's ETL language. The basic transformation exists of transforming a BCTT model into a UPPAAL Network of Timed Automata (NTA). This transformation consists of two parts; the first part sets up the templates in UPPAAL including the required channels and variables. The second part creates the system that represents the taskmodel.

In the setup phase of the transformation, every UPPAAL template that we defined in Section 4.3 is created. We can consider this boilerplate code as it is the same for every transformation. The templates are later referred to when we define the UPPAAL system.

The actual transformation from BCTT model to UPPAAL system happens in phase two.

```
// Part of CTTXML2CTT.etl
rule CTTXML2CTT
    transform cttxml : CTTXML!t_TaskModel
    to ctt : CTT!TaskModel {

    ctt.name = cttxml.a_NameTaskModelID;

    var child : CTT!Task = cttxml.e_task.recursiveXMLTeardown(ctt);
    child.TaskToBinary(ctt);

    ctt.root = child;
}
```

Listing 4: Example ETL file to transform CTTE output files (XML) into PCTT models. - this rule looks for all CTTXML!t_TaskModel items in the XML file and creates a BCTT!TaskModel for it. The code in the rule is executed for all transformations.

For every task in the BCTT model, a UPPAAL template instantiation is created and added to the UPPAAL system. Listing 6 shows how a template declaration in UPPAAL is created for a BCTT interaction task. The transformation rule calls the `createTemplateDeclartion` operation which creates a declaration for the correct template based on the leaf task type or non-leaf task operator.

When the conversion from BCTT models to UPPAAL systems is done, the UPPAAL system (which conforms to the UPPAAL metamodel) can be serialized and written to a UPPAAL file. The generated UPPAAL file is now available to the user as a UPPAAL XML file.

## 4.5. Creating the Query Generator Tool

In the previous sections we have written about the internals of our tool. We have explained the metamodels for both PCTT and UPPAAL and how we can transform models that conform to the PCTT metamodel into UPPAAL models. To make sure the transformations are correct, we have added automatic validation to the transformations. However, we are not yet capable of actually using the tool. For this we will need a *user interface* (UI) that allows the user to select a CTTE made task model and perform queries on it in a user friendly manner.

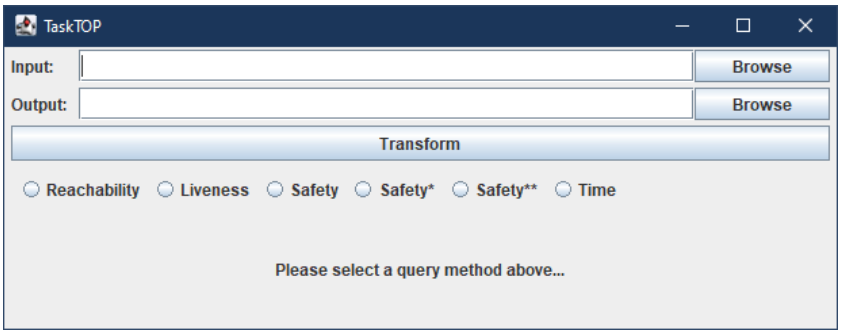



Figure 4.13: The TaskTOP tool.

```
// Part of CTTXML2CTT.etl
operation CTTXML!t_Task recursiveXMLTeardown(taskModel : CTT!TaskModel)
 ↪  : CTT!Task{
    var subtask = self.e_subtask;
    var children = subtask<>null ? subtask.c_task : null;

    // create abstraction, interaction, application or user task
    var result = self.getTaskImplementation();
    result.id   = self.a_Identifier;
    result.name = self.e_name != null ? self.e_name.text : "";
    result.iterative = self.a_Iterative.asBoolean();

    var time_performance = self.getTimePerformance();
    result.min = time_performance.get("min").round();
    result.max = time_performance.get("max").round();

    taskModel.tasks.add(result);

    if(children != null and children.size() > 0) {

        var leftChild = children[0];
        result.operator = leftChild.getTemporalOperator();

            for(c in children) {
                if(c.isTypeOf(t_task)) {
                    var child : CTT!Task =
                     ↪  c.recursiveXMLTeardown(taskModel);
                    result.subtasks.add(child);
                }
            }
    }

    if(self.a_Optional.asBoolean()) {
        return handleOptional(result, taskModel);
    }

    return result;
}
```

Listing 5: Operation that recursively walks through the XML task model tree and converts it to BCTT. For every XML task, it creates a CTT task en copies the relevant information over. If the XML task has subtasks (children), it will store the TempOp of the first child and then call the recursive operation on all of the subtasks. The operation also handles the 'special' optional tasks.

```
// Part of CTT2UPPAAL.etl
rule interactionTask2Template
transform task : CTT!InteractionTask
to ret : List {
    task.createTemplateDeclaration(ret, 1);
}

operation CTT!Task createTemplateDeclaration(ret : List, type : Integer)
 ↪  {
    var index = taskIndex;
    taskIndex++;

    var decl = new Uppaal!TemplateDeclaration();
    decl.declaredTemplate = new Uppaal!RedefinedTemplate();
    decl.declaredTemplate.name = "t_" + self.id.asString();
    if(self.operator == null) {
            // so leaf task...
            decl.declaredTemplate.referredTemplate =
             ↪  getTemplateByTaskType(type);
    } else {
            // so non-leaf task...
            decl.declaredTemplate.referredTemplate =
             ↪  getTemplateByOperatorType(self.operator);
    }

    decl.argument.add(createLiteralExpression(index));
    decl.argument.add(createLiteralExpression(self.iterative));

    if(self.operator != null) {
        decl.argument.add(createLiteralExpression(
         ↪  self.subtasks.get(0).equivalent().get(1) ));
        decl.argument.add(createLiteralExpression(
         ↪  self.subtasks.get(1).equivalent().get(1) ));
    } else {
        decl.argument.add(createLiteralExpression(self.min));
        decl.argument.add(createLiteralExpression(self.max));
    }

    ret.add(decl);
    ret.add(index);
}
```

Listing 6: Example ETL rule that creates template declarations for all interaction tasks in the BCTT model.

Figure 4.13 shows the UI for our tool called TaskTop[2]. The tool reuses parts of the ATTop tool [Kumar et al., 2018] and shares the same vertical layout structure. TaskTop has a very simple UI that is laid out in a top-to-bottom manner. The user starts at the top by selecting the CTTE file that contains the PCTT task model. An output file can also be specified but is not mandatory. The output file contains (after transformation) the UPPAAL version of the PCTT task model and can be used to open the model in UPPAAL manually. Note that, when no output file is specified, an output file will still be created but with a default name.

When the correct file is specified, the user must click the *Transform* button to start the model transformation. If the transformation was successful, the button turns green and the user can proceed to the next step; performing a query on the task model. If the transformation failed for some reason, the button will turn red and the error message is displayed in a popup window.

### 4.5.1. Queries

When the transformation of the task model was successful, the user can perform queries on it. Under the hood, UPPAAL is used to query the model using UPPAAL queries. UPPAAL queries are not always very user friendly and in most cases, users that know task models do not necessarily know UPPAAL and its queries. To overcome this, we have created a 'user friendly' way of creating queries and let users make queries by letting them select the type of query and the tasks that need querying.

#### Reachability query

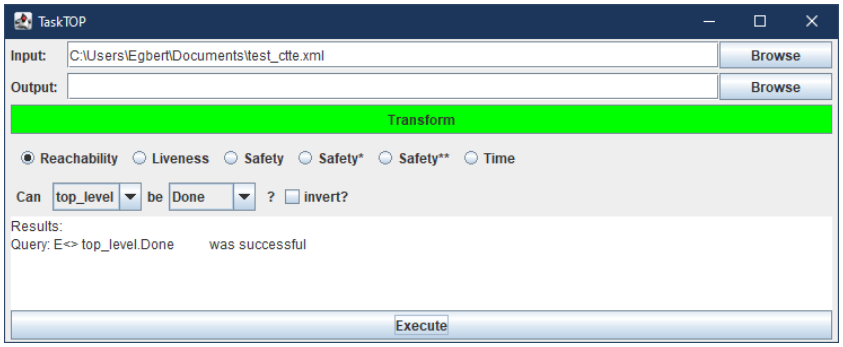



Figure 4.14: TaskTOP - Performing a reachability query.

The first type of query that we can verify using our tool is the *Reachability* query. A reachability query verifies that a certain task (or tasks) can be reached. Figure 4.14 shows how we can enter such a reachability query. We only need to select the task and the state of that task (Disabled, Done, etc...) and click the *Execute* button. The tool will create the UPPAAL equivalent query for us and perform it on the model. When the querying is done, the tool shows us if the query was successful or not. Note that the query can also be inverted and be changed into a query that tests if a task state cannot be reached. "Can top_level **not** be Done?" is an example of such an inverted reachability query.

#### Liveness query

The second type of query that we can verify using our tool is the *Liveness* query. A liveness query is used to verify that a task will eventually always be in a certain state. In Figure 4.15,

[2]Source code can be found on https://github.com/egbertpostma/TaskTOP

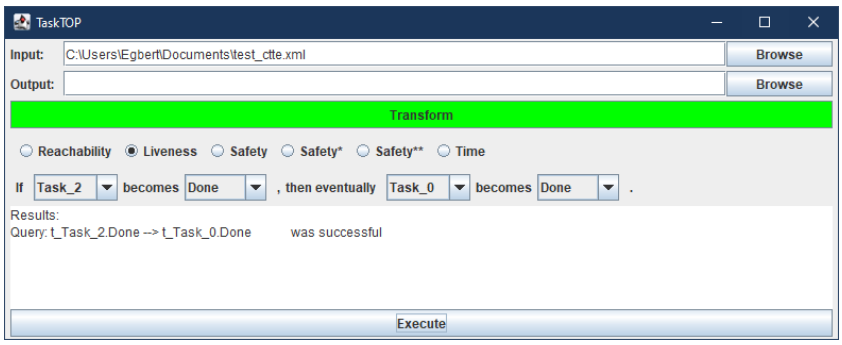



Figure 4.15: TaskTOP - Performing a liveness query.

we test that if Task_2 is done, then Task_1 is also eventually done. The result of the query is shown in the result section below the query.

### SAFETY QUERY

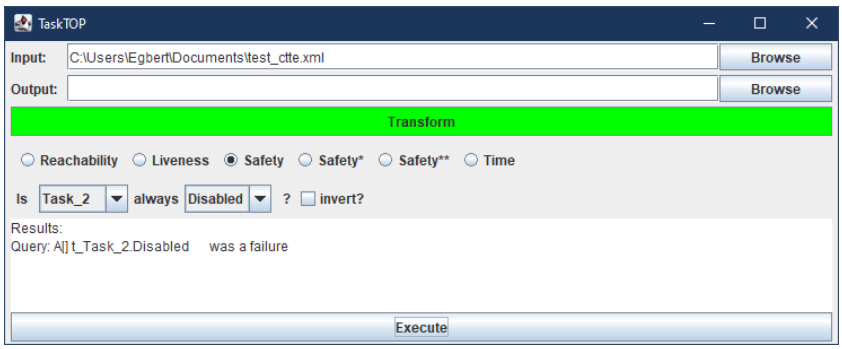



Figure 4.16: TaskTOP - Performing a safety query.

The third type of query that we can verify is the *Safety* query. The safety query is used to check whether a task is always or never in a certain state. The default setting for the query is *always*, but the query can be inverted so its setting becomes *never*.

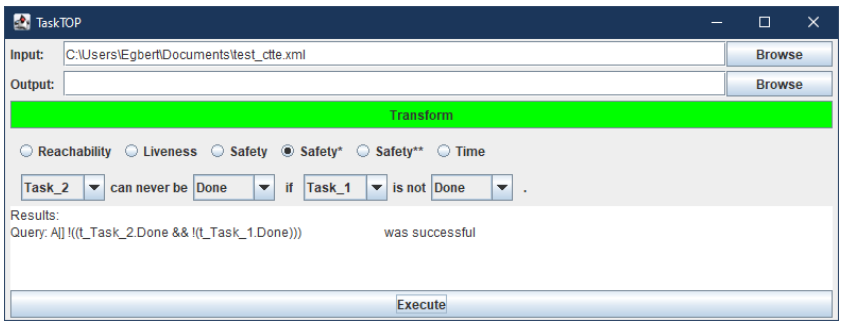



Figure 4.17: TaskTOP - Performing a safety query.

The simple *Safety* query is not always useful. It can be used to verify, for instance, that an error task can never be done. Instead, it would be more useful to test whether a combination of tasks can never occur. That is why we have also included the *Safety** query. This special safety query can be used to verify that certain combinations of tasks can not happen. This query is also entered in a human readable manner. We first enter the task

of which the safety must be ensured, then we enter the task that provides the safety. In the example in Figure 4.17, the safety of a *SequentialEnabling* operator is tested, i.e. Task_2 can never be done if Task_1 is not done.

# 5

# VALIDATION

In this chapter, we will show the methods we used to validate our work. We start with the validation of the UPPAAL representations that we have created by verifying properties of them in UPPAAL. After that, we show how we validated the model transformations using Epsilon's Validation Language. Lastly, we show the validation of the Query Generator Tool we have created. Here, we start with a simple task model and eventually perform model checking on a more complex task model.

## 5.1. VALIDATION OF UPPAAL REPRESENTATIONS

The first step in the validation process is to validate that our representations of the CTT elements in UPPAAL are correct. We will use UPPAAL to do this for us. We can manually create systems in UPPAAL that can be used to perform queries on. In this section, we will show the validations for some of the representations like the leaf tasks, non-leaf tasks, optional tasks and iterative tasks. Every validation consists of a UPPAAL system accompanied with a number of queries to perform on the system.

### LEAF TASK VALIDATION

Leaf tasks are the tasks that are at the bottom of the task tree. A leaf task is also the task that is literally executed, i.e. it can be started by the system or the user. A leaf task can be guarded with time constraints that specify its minimum and maximum duration. A setup to test the leaf task implementation in UPPAAL is shown in Listing 7. There are four different leaf tasks that can be defined; one with no time constraints, one with a minimum time constraint, one with a maximum time constraint and one with both a minimum and maximum time constraint. To perform the tests, we need to assign the appropriate `min` and `max` values (`-1` means not set).

Property 1 of Listing 8 verifies that the task can be done and time has not increased, i.e. the task was done instantly. This can be valid because when the task is in the *Done* state, time is not allowed to run (`time' == 0`). The second property (2) of Listing 8 verifies that a task can be running indefinitely and never finish, i.e. there is no maximum duration. Queries 3 and 4 of Listing 8 validate that a task takes at least two time units. Query 3 states that when the task is done, its time value is always greater than or equal to 2. Query 4 states that it is never possible for a task to be done when its time value is less than 2. Queries 5 and 6 of Listing 8 validate that a task takes at maximum five time units. Also here, Query 5

```
top_level   = TopLevel(0);
task_0      = InteractionTask(0, false, <min>, <max>);
//task_0      = ApplicationTask(0, false, <min>, <max>);

system top_level, task_0;
```

Listing 7: A UPPAAL system to test leaf tasks.

states that when a task is done, its time value is always smaller than or equal to 5 and Query 6 states that it is never possible for the task to be done with a time value that is greater than 5.

```
(1) E<> task_0.time == 0 && task_0.Done          // min: -1, max: -1
(2) (task_0.Running --> task_0.Done) == false    // min: -1, max: -1
(3) A[] (task_0.Done imply task_0.time >= 2)     // min: 2,  max: 5
(4) A[] !(task_0.Done && task_0.time < 2)        // min: 2,  max: 5
(5) A[] (task_0.Done imply task_0.time <= 5)     // min: 2,  max: 5
(6) A[] !(task_0.Done && task_0.time > 5)        // min: 2,  max: 5
```

Listing 8: Verification queries for leaf tasks.

The previous tests were mainly focused on *Interaction tasks* but should also be true for *Application tasks*. To also test the specifics of both the *Interaction tasks* and *Application tasks*, the queries in Listing 9 are used. The first Query (1) states that whenever an application task becomes enabled, it always eventually becomes active too. This does not hold for interaction tasks, as is stated by Query 2. Here, it is verified that there exists a path in which the task never becomes active. Query 3 states that, in case of an application task that has a maximum time specified, it always becomes done after it has been enabled. Query 1-alt and 3-alt can also be used to verify that an application task with a maximum time always becomes active and/or done because of the test setup as the top level task is set to always immediately enable its subtasks.

```
// Application task, min: -1, max: -1
  (1) task_0.Enabled --> task_0.Active
// Interaction task, min: -1, max: -1
  (2) E[] !task_0.Active
// Application task, min: -1, max: 5
  (3) task_0.Enabled --> task_0.Done

  (1-alt) A<> task_0.Active
  (3-alt) A<> task_0.Done
```

Listing 9: Verification queries for Application tasks and Interactive tasks.

### NON-LEAF TASK VALIDATION

Non-leaf tasks cannot be validated without the use of leaf tasks as they react to their subtasks. Therefore, to test the non-leaf tasks, we will create UPPAAL systems that consist of the non-leaf task and has two leaf subtasks assigned to it. The leaf tasks will be of type *Application task* so they are guaranteed to be executed.

**SequentialEnabling task validation** The CTT model in Figure 5.1 is an example of a simple task model that can be used to validate the *SequentialEnabling task* representation in UPPAAL. The system of this task model can be seen in Listing 10.

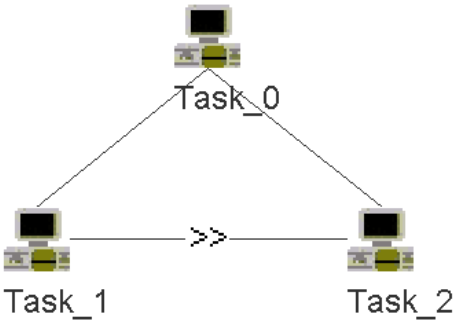



Figure 5.1: Simple CTT example of a SequentialEnabling task

```
top_level = TopLevel(0);
task_0    = SequentialEnabling(0, false, 1, 2);
task_1    = ApplicationTask(1, false, -1, -1);
task_2    = ApplicationTask(2, false, -1, -1);

system top_level, task_0, task_1, task_2;
```

Listing 10: A UPPAAL system to test sequential enabling tasks.

To test the *SequentialEnabling task*, we check for different properties that should be true for this task model. First, if *Task_0* is disabled, both *Task_1* and *Task_2* should become disabled too. The query with which we test this is described in Listing 11 Query 1. The query tests that, whenever *Task_0* is disabled, both its subtasks are also disabled.

```
(1) Task_0.Disabled --> Task_1.Disabled && Task_2.Disabled
(2) Task_2.Done --> Task_0.Done
(3) A[] !(!Task_1.Done && Task_2.Done)
(4) E<> Task_1.Done && !Task_2.Done
(5) Task_1.Done --> !Task_2.Disabled
```

Listing 11: Verification queries for the SequentialEnabling task.

The next property of the SequentialEnabling task is that, after the right subtask is done, the task is also done. We can verify this with Query 2 in Listing 11. The query implies that if *Task_2* is done, *Task_0* always becomes done too.

To ensure that the SequentialEnabling task is actually sequential, we can use reachability and safety properties. Query 3 of Listing 11 is used to validate that there is no state in which *Task_1* is not done and *Task_2* is done. This is an essential property for the *SequentialEnabling task* as it requires the left subtask to be done in order for the second subtask to be enabled. Query 4 of Listing 11 verifies that there is such a path in which the first subtask is done and the second is not done. The last query (5) verifies that, whenever *Task_1* is done, *Task_2* will no longer be disabled anymore.

### ITERATIVE TASK VALIDATION

An iterative task is a task that, once it finishes, automatically starts over again. A consequence of this is that an iterative task can never become done. Also, an iterative task can only be stopped by disabling it. To test if an iterative task is actually iterative, we can test the property that it can never become done.

In order to test the iterativity of a task in UPPAAL, we need to create a setup that includes an iterative task. Listing 12 describes a UPPAAL system with only one application (or interaction) leaf task that is configured to be iterative.

```
top_level = TopLevel(0);
task_0    = ApplicationTask(0, true, -1, -1);

system top_level, task_0;
```

Listing 12: A UPPAAL system to test iterative leaf tasks.

The query that we can use to verify the property is Query 1 of Listing 13. The query states that there exists no state in which *task_0* is done. The query evaluates to `true` and so this property is valid.

```
(1) A[] !task_0.Done
```

Listing 13: Verification queries for an iterative task.

### OPTIONAL TASK VALIDATION

An optional task is a task that may or may not be executed in order for its parent task to become done. We cannot define an optional task in UPPAAL straightaway because there is no direct representation of it. Instead, we must replace the optional task with a choice task that has the optional task as its left subtask and a null-task as its right subtask. The configuration is shown in Listing 14. The original optional task (*task_0*) has become a subtask of the new choice task (*task_0_opt*). The second subtask of the new choice task is the null task (*task_0_null*).

To test the optional task, we can check if there is a state possible in which the top-level task is done and the optional task is not. A query that would test this is Query 1 of Listing 15. For completeness, we also test that it is possible for the optional task to be actually done in order for its parent to become done (Query 2 of Listing 15).

```
top_level    = TopLevel(0);
task_0_opt   = Choice(0, false, 1, 2);
task_0       = ApplicationTask(1, false, -1, -1);
task_0_null  = NullTask(2);

system top_level, task_0, task_0_opt, task_0_null;
```

Listing 14: A UPPAAL system to test optional tasks.

```
(1) E<> top_level.Done && !task_0.Done
(2) E<> top_level.Done && task_0.Done
```

Listing 15: Verification queries for an optional task.

## 5.2. Validation of BCTT to UPPAAL transformations

In Section 4.4 we have explained how we are converting BCTT models in to UPPAAL models. We used Epsilon's ETL language for performing the transformations. To make sure the models that we are converting are correct, we used Epsilon's EVL language. Because we use the EVL validations, we implicitly show that our conversions are valid. Of course, the correctness of the validation is based on the validity of the rules we have set.

The rules we have used in the EVL validations are not arbitrary. They are based on the definitions we have discussed in Chapter 2. Definition 4 states, for instance, that, in a well formed PCTT, non-leaf tasks consist of a minimum of two subtasks and that a task model has only one *Goal* (root task). By verifying these rules at runtime, we can make sure that our models are always correct.

For the validation of the transformations, we use the same method. We use Epsilon's EVL language to validate the transformations. The rules that we use here are also based on the definitions in Chapter 2. Definition 4 namely states that, in a well formed PCTT, every non-leaf task should have a *TempOp* assigned to it and every leaf task should have a *TP* assigned to it. Because of this, we can create an EVL rule that states that there should be a template instance in UPPAAL for every task (leaf or non-leaf) in PCTT. This way, the transformations are validated at runtime which ensures that the generated models are correct.

## 5.3. Validation of the Query Generator Tool

To validate the QGT we cannot make use of Epsilon's EVL language. Instead, we validate the QGT manually using some simple and more complex task models which we transform using the QGT and then perform queries on of which we already know the results. We use CTTE to create the task models and, where necessary, prioritize them. First, we will examine the task model manually and also question the task model for certain properties by hand. We will then use the QGT to do the transformation and the execution of the queries and match the output of the tool to our manually obtained answers.

### 5.3.1. Validating a simple task model

If we take for instance the task model of Figure 5.2, we can question it that in order to click submit, we first need to enter the first and last name. The query would look something like

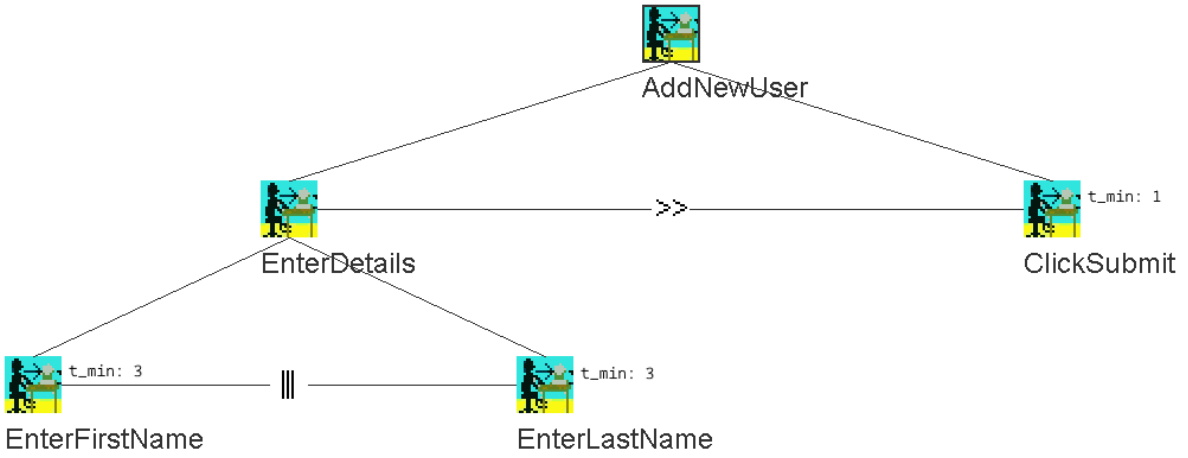



Figure 5.2: CTT example - Add new user task (with time performance values).

"if `ClickSubmit` is done, then `EnterDetails` is done". To test this in QGT, we can enter the query as shown in Figure 5.3. The QGT also shows that the query holds for this task model.

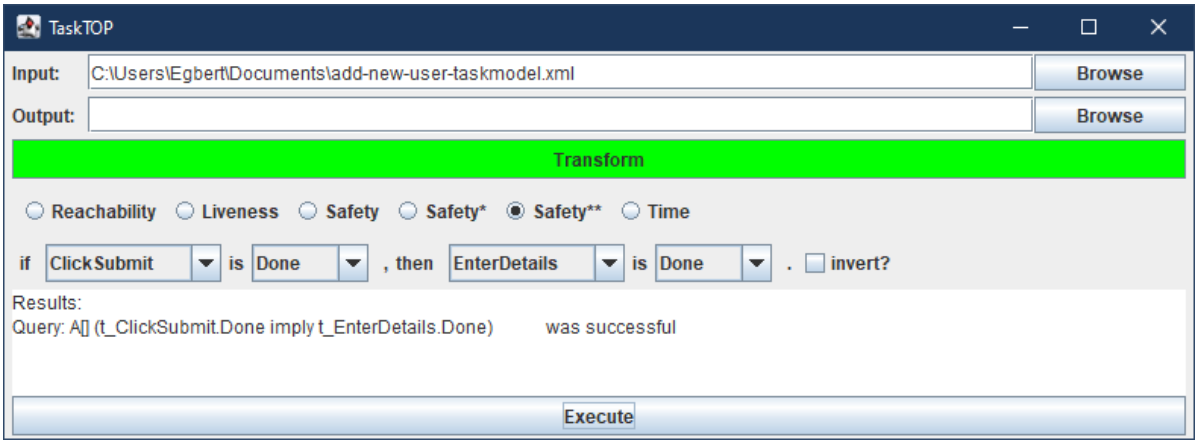



Figure 5.3: CTT example - Add new user task query in TaskTOP QGT

When we enrich the task model with time performance values for the leaf tasks, we can also validate queries like "can this task be finished within ... time units". For instance, if both the *EnterFirstName* and *EnterLastName* tasks would take at minimum three time units to finish and the *ClickSubmit* task at minimum one time unit, then the total minimum time of the task model would be four time units. We can verify this by asking the tool to check if the entire task model (top level task) can be finished within four time units, or "can task model be done and elapsed time of task model be smaller than four?". This question results in a failure as this should not be possible. On the other hand, one could perform the following query on the system: "if task model is done, then elapsed time of task model is always greater than or equal to four", which does return successful. This query is entered in the QGT as shown in Figure 5.4

### 5.3.2. Validation of a more complex task model

To verify that our tool is not only working correctly with simple task models but also with the more complex ones, we test it using the task model in Figure 5.5. The task model is a

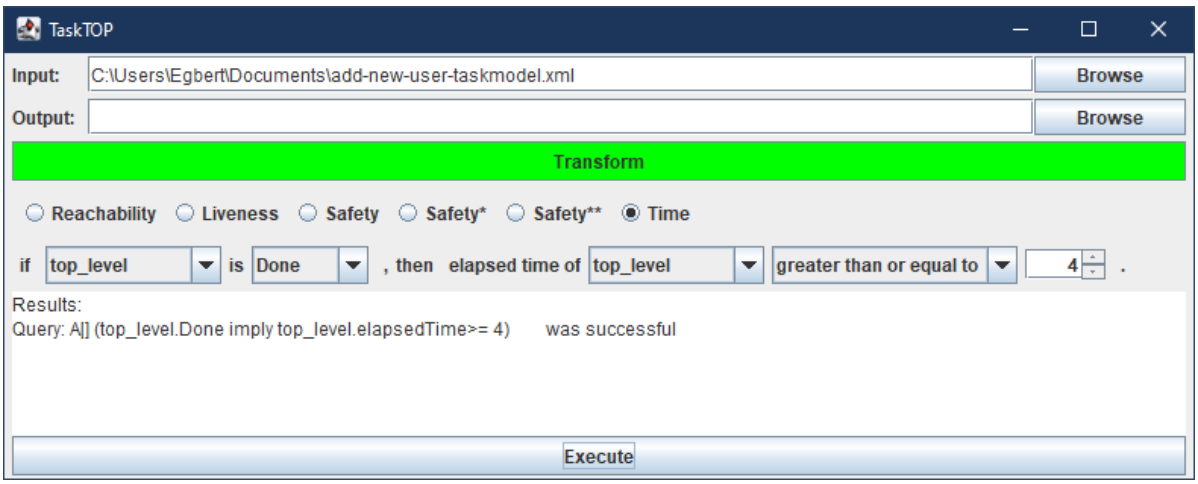



Figure 5.4: CTT example - Add new user task time query in TaskTOP QGT

simplified model of an ATM that still has plenty of tasks in it for us to test. The task that is modeled here is *Access ATM*. The task is divided in three subtasks; *Enable Access*, *Access* and *Close Access*. In order to access the ATM, a user should first enable access by inserting his card and submitting his password. Only then, the user can access the ATM and, for instance, withdraw or deposit cash. Closing the access to the ATM can be done at any given moment after access was enabled.

We have chosen the task model in Figure 5.5 because it is complex but yet simple enough for it to be manually validated too. This way, we can use it to validate the tool and compare the results with our expectations.

In case of an ATM, it is of course very important that a user cannot withdraw cash without being granted access. Therefore, it is mandatory for a user to: *first* enable access and *then* withdraw cash. In the task model we can see that this is taken care of by the *SequentialEnabling operator* between the tasks *EnableAccess* and *Access**. All the subtasks of *Access** only become enabled after *EnableAccess* is done. This means that, in order for the *WithdrawCash* task to become done, the *EnableAccess* task must be done too. Otherwise said: "if *WithdrawCash* is done, then *EnableAccess* is done", which is an implication. We can verify this in UPPAAL through the following query:

```
A[] (WithdrawCash.done imply EnableAccess.done)
```

However, using our tool, we do not need to worry about UPPAAL queries. Instead, we can enter the implication in the QGT as a *safety query* and verify that the implication holds for the task model. In Figure 5.6, the result of the query can be seen. The ATM example task model is loaded into the tool and transformed into a UPPAAL model. Then, the *safety** *query* method is selected and configured to verify that cash withdrawal is only possible after the user was enabled access. Executing the query shows that it is successful for the given task model.

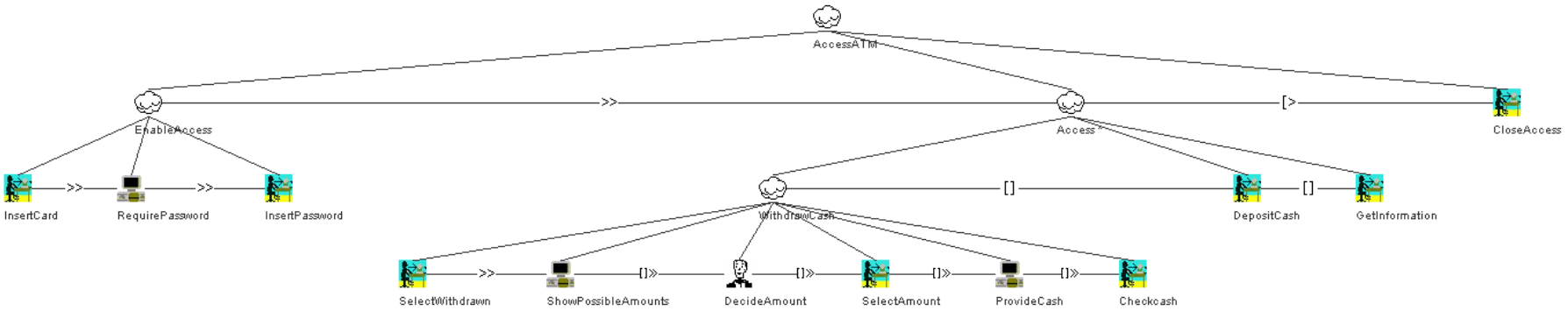



Figure 5.5: CTT example - Task model of an ATM machine.

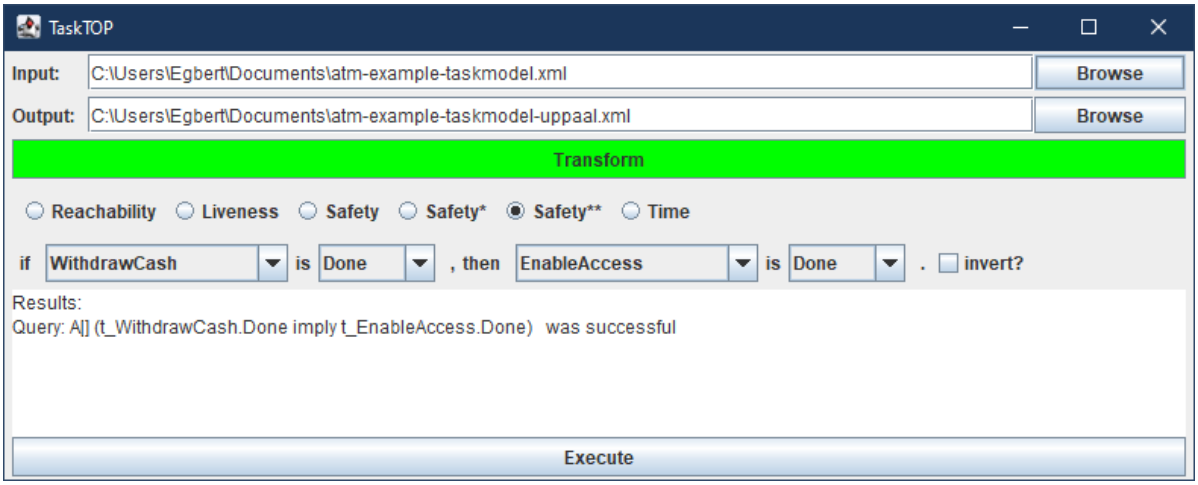



Figure 5.6: TaskTOP - ATM example query.

# 6

# CONCLUSION

In this master thesis, we have created a tool with which one can perform model checking on task models. Existing tools that work with task models are incapable of verifying properties of those task models automatically. Instead, the user must manually simulate the task model step-by-step to find out if properties hold. TaskTop, the tool that we have created, *can* verify properties of task models automatically. A user can load the task model to test and enter queries that verify the properties that need verification in a user friendly way.

TaskTop depends on UPPAAL to perform the model checking. UPPAAL is a tool that can verify properties of Networks of Timed Automata (NTA) using queries. Because task models are not timed automata, we needed to find a way of converting task models into timed automata. This brings us to the main research question:

*"How can MDE assist in checking Task Models with UPPAAL?"*

In order to answer this question, we first looked at another similar tool (ATTop) that allows a user to perform model checking on Attack Trees using UPPAAL [Kumar et al., 2018]. As Attack Trees have similarities with task models, we performed an in-depth comparison between the two. Based on the comparison, we have reused parts of the ATTop tool for our TaskTop tool.

The next step into answering the research question was to design a metamodel for task models. To transform models from one domain to another, we have made use of Model Driven Engineering (MDE) and used the Eclipse Modeling Framework (EMF) in combination with Epsilon for performing the model transformations. EMF is able to do model transformations based on transformation definitions. Such a transformation definition *knows* how to transform elements from one metamodel to elements in the other metamodel. TaskTop reads in the task model and converts it into a UPPAAL model after which the UPPAAL model is sent to UPPAAL to perform queries on.

To make TaskTop *the* tool to use, we needed to ensure that using our tool is simple. A user that has knowledge of task models does not necessarily has knowledge of UPPAAL and its way of writing queries. For this, we created a user friendly manner for the user to enter queries with which he or she can verify properties on the task model. The queries are written in plain English and are converted into Timed Computation Tree Logic (TCTL) which is the notation that UPPAAL uses for its queries. TaskTop sends the queries to UPPAAL which,

in turn, executes the queries. The results of the queries are then sent back to TaskTop and are displayed to the user.

With this thesis, we have shown that we can create a tool that allows users to perform model checking on task models using UPPAAL as a model checker. For this reason, we can call our study a success. In the next section we will discuss our study and describe the weaknesses and strengths of our study.

## 6.1. Discussion

In this section, we are discussing the weaknesses and strengths of our study. As certain aspects of task models were not used in our tool, we explain here why they were not used and how that strengthened our research. There are also aspects that might have strengthened our study but were left out for now, these aspects can be considered weaknesses of the performed research.

### Enabled Task Sets

One way of converting a task model to a state transition network is to make use of enabled task sets (ETSs). An ETS is a set of tasks that can be executed at the same time, thus, that are enabled at the same time. An ETS represents a state of the task model in which certain tasks are enabled. When a task is executed, the state of the model is updated to the next set of enabled tasks. ETSs are explained in Chapter 2 Section 2.2.4 and an example of an ETS can be found in Figure 2.7.

Using ETSs to convert task models to STNs is quick and simple method and can be done using the CTTE tool. It generates STNs that can be perfectly used for reachability analysis for example. If one would like to know whether or not two tasks can be enabled at the same time for instance, one could simply look for the ETS in which both tasks are enabled. Another possibility is to verify if a task can still be executed if another task has been executed before.

Besides the fact that relatively simple STNs are generated which can be easily model checked, the method also has some downsides. For example, in the resulting STN of Figure 2.7, there is no reference to $Task\_1$. In fact, all non-leaf tasks are omitted in STNs based on ETSs. Verifying properties of the task model that relate to $Task\_1$ is therefore not possible when using the ETS method.

Concluding the topic of ETSs, ETSs can be useful to perform quick reachability properties of a task model but it also hides non-leaf tasks of which properties cannot be checked. For this reason, we have chosen *not* to make use of ETSs in our tool.

### Create TaskTop from scratch

Because our tool is supposed to do something quite similar to the ATTop tool, it would have made perfect sense to modify the tool so it could *read in* task models and leaf the rest mostly as is. This was actually what has been tried in the beginning of this research. However, we quite quickly found out that getting the ATTop tool to compile in our Eclipse environment was very hard.

The code of ATTop contained, according to today's standards, a lot of deprecated methods and used versions of libraries that either, were not available anymore or did not work with the latest java versions anymore. That is when we decided to create a new Eclipse project from scratch and importing (and updating) the UPPAAL metamodels libraries from

there. The choice of recreating the tool from scratch did, in the end, *not* consume a lot of extra time as the existing tool was used as a reference.

### Comparison with other tools

One of the weaknesses of our work is that we have not compared our tool against other existing tools. The main reason for this is that there are no actual tools available that can perform model checking on CTT task models, especially using UPPAAL. The tools that do exist, for instance CTTE, can be used to perform simulations on task models. Simulating task models *can* be considered as manual model checking, but comparing that to our tool seemed not fair as we perform automatic model checking. Because of this, we cannot say that our tool outperforms other task model checking tools other than that we *can* perform model checking on task models.

### Incomplete implementation of CTT notation

In our work, we have implemented the most basic features of the CTT notation. This means that we have also left features out of scope. The tool we have created, TaskTop, is able to execute queries on task models that are based on the CTT notation. However, we have not implemented all features of CTT yet. We do support all temporal operators as well as optional and iterative tasks. We also support the use of time performances, but we do not support the use of objects in CTT and Cooperative CTT. These are features that might be studied in possible future work, which leads us to the following section.

## 6.2. Future work

In this section we will propose possible future work based on the results of our work. Topics that might need future work are the usage of *Objects* in CTT, adding support for *Cooperative CTT (CCTT)* and showing UPPAAL query results in the actual task model.

### Objects

Objects in CTT can be used to assign pre- and/or post-conditions to tasks. Tasks cannot be executed if their pre-condition is not satisfied and cannot become done if their post-condition is not satisfied. How CTT handles objects is not very well documented, but future work might be able to clarify this so it can be added to TaskTop.

### CCTT

Another feature of CTT that we currently do not support is Cooperative ConcurTaskTrees (CCTT). CCTT enables users to create multiple task models and let them work together. An example of such a cooperative task model is a task model for a pilot that needs to land an airplane and a task model of the tower operator that needs to guide the airplane to the correct landing strip. CCTT allows the *cooperation* between those two task models. To support this feature in TaskTop, additional research needs to be done.

### Show UPPAAL results in task model

To help the user of TaskTop even further, more research could be done on how to get the query results back to the task models. If a user could see, for example as a trace in the task model, why a certain property does not hold, the user can *pinpoint* more easily where the problem exists and fix it.

Another feature that could be studied more and maybe extends the previous idea is automatic property testing. For example, when all tasks are *automatically* verified for reachability, then the user could get a visual representation of all *non-reachable* tasks.

## 6.3. REFLECTION

When I started with the research project, I was very encouraged and excited to get to work. Besides being a student, I am also a full-time employee at the Dutch Ministry of Defence as a software developer. Luckily, I was allowed one full workday per week to spend on studying. Still, I found that working on a masters thesis, which includes creating software, and working four days a week also as a software developer is hard sometimes. Because the full day I had for studying was not enough, I also spend most evenings on studying as well. Finding the motivation to sit behind a laptop *again* after already working with the computer the whole day was not always easy. Luckily, I did manage to get it done and I can look back at an educational time, not only software-wise but also time-management-wise.

On the matter of the graduation project and writing this thesis, I did get a lot from the contact moments with my supervisor Dr. Stefano Schivo. The feedback that I got when submitting draft versions of my thesis to him was very useful and educational. He not only wrote comments on what I did wrong or should improve, but also complimented me on the good parts. This was a real motivator and encouraged me to persevere.

In the end, I have found this period of graduating to be well worth it and I have learned a lot from it. On the question if I would do it again; well, maybe in the future. For now, I am happy that I managed to pull it off and glad that I have made the choice of getting my masters degree in software engineering.

# BIBLIOGRAPHY


Rajeev Alur and David Dill. The theory of timed automata. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 600 LNCS, pages 45–73. 1992. ISBN 9783540555643. doi: 10.1007/BFb0031987. 14

Florian Arnold, Axel Belinfante, Freark Van Der Berg, Dennis Guck, and Mariëlle Stoelinga. Dftcalc: A tool for efficient fault tree analysis. volume 8153 LNCS, pages 293–301. Springer, Berlin, Heidelberg, 2013. ISBN 9783642407925. doi: 10.1007/978-3-642-40793-2_27. URL https://link.springer.com/chapter/10.1007/978-3-642-40793-2_27. 23

Mickaël Baron and Patrick Girard. SUIDT : A task model based GUI-Builder. *TAMODIA : Task MOdels and DIAgrams for user interface design*, 1(January 2002):64–71, 2002. 1

Gerd Behrmann, Alexandre David, and Kim G Larsen. Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3185, pages 200–236. 2004. ISBN 978-3-540-30080-9. doi: 10.1007/978-3-540-30080-9_7. 15, 17, 18

Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *ACM SIGMETRICS Performance Evaluation Review*, 32(4):34–40, mar 2005. ISSN 0163-5999. doi: 10.1145/1059816.1059823. 18

Gerd Behrmann, Alexandre David, Kim Larsen, John Håkansson, Paul Pettersson, Wang yi, and Martijn Hendriks. Uppaal 4.0. pages 125–126, 01 2006. doi: 10.1109/QEST.2006.59. 2

Stuart K Card, Thomas P Moran, and Allen Newell. *The psychology of human-computer interaction*. L. Erlbaum Associates Inc., 1983. ISBN 978-0898598599. 1, 32

Olga Gadyatskaya, Ravi Jhawar, Piotr Kordy, Karim Lounis, Sjouke Mauw, and Rolando Trujillo-Rasua. Attack Trees for Practical Security Assessment: Ranking of Attack Scenarios with ADTool 2.0. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9826 LNCS, pages 159–162. Springer Verlag, 2016. ISBN 9783319434247. doi: 10.1007/978-3-319-43425-4{_}10. URL http://link.springer.com/10.1007/978-3-319-43425-4_10. 23

Christopher Gerking. *Transparent UPPAAL-based Verification of MechatronicUML Models*. PhD thesis, University of Paderborn, 2013. 24

ISO 8807:1989. Information processing systems — open systems interconnection — lotos — a formal description technique based on the temporal ordering of observational behaviour. Standard, International Organization for Standardization, Geneva, CH, February 1989. 6

Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. Foundations of attack-defense trees. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6561 LNCS, pages 80–95. Springer, Berlin, Heidelberg, 2011. ISBN 9783642197505. doi: 10.1007/978-3-642-19751-2_6. URL https://link.springer.com/chapter/10.1007/978-3-642-19751-2_6. 22

Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. DAG-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer Science Review*, 13-14(C):1–38, nov 2014. ISSN 15740137. doi: 10.1016/j.cosrev.2014.07.001. URL https://linkinghub.elsevier.com/retrieve/pii/S1574013714000100. 22, 24

Rajesh Kumar, Enno Ruijters, and Mariëlle Stoelinga. Quantitative Attack Tree Analysis via Priced Timed Automata. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9268, pages 156–171. 2015. ISBN 9783319229744. doi: 10.1007/978-3-319-22975-1_11. URL http://link.springer.com/10.1007/978-3-319-22975-1_11. 3

Rajesh Kumar, Stefano Schivo, Enno Ruijters, Buğra Mehmet Yildiz, David Huistra, Jacco Brandt, Arend Rensink, and Mariëlle Stoelinga. Effective Analysis of Attack Trees: A Model-Driven Approach. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10802 LNCS, pages 56–73. 2018. ISBN 9783319893624. doi: 10.1007/978-3-319-89363-1{_}4. URL http://link.springer.com/10.1007/978-3-319-89363-1_4. 3, 23, 28, 48, 60

Quentin Limbourg and Jean Vanderdonckt. Comparing task models for user interface design. In *The handbook of task analysis for human-computer interaction*, volume 6, page 135–154. Lawrence Erlbaum Associates, 2003. URL http://hdl.handle.net/2078/17948. 1, 4

Sjouke Mauw and Martijn Oostdijk. Foundations of Attack Trees. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3935 LNCS, pages 186–198. 2006. ISBN 3540333541. doi: 10.1007/11734727_17. URL http://link.springer.com/10.1007/11734727_17. 22, 31

F. Paternò, C. Mancini, and S. Meniconi. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In *Human-Computer Interaction INTERACT '97*, pages 362–369. Springer US, 1997. doi: 10.1007/978-0-387-35175-9_58. URL http://link.springer.com/10.1007/978-0-387-35175-9_58. 1, 4, 8, 32

Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*, volume 10 of *Applied Computing*. Springer London, London, 2000. ISBN 978-1-85233-155-9. doi: 10.1007/978-1-4471-0445-2. URL http://link.springer.com/10.1007/978-1-4471-0445-2. 10

Fabio Paternò, Giulio Mori, and Riccardo Galiberti. CTTE: An environment for analysis and development of task models of cooperative applications. In *Conference on Human Factors in Computing Systems - Proceedings*, pages 21–22, 2001. ISBN 1581133405. doi: 10.1145/634067.634084. URL http://giove.cnuce.cnr.it/guitare.html. 2, 22

Fabio Paternò, Carmen Santoro, Lucio Davide Spano, and Dave Raggett. MBUI - Task Models. *W3C*, (April), 2014. URL https://www.w3.org/TR/task-models/. 22, 27, 28

Alberto Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, oct 2015. ISSN 14778424. doi: 10.1016/j.cl.2015.06.001. 2, 19

Stefano Schivo, Buğra M. Yildiz, Enno Ruijters, Christopher Gerking, Rajesh Kumar, Stefan Dziwok, Arend Rensink, and Mariëlle Stoelinga. How to Efficiently Build a Front-End Tool for UPPAAL: A Model-Driven Approach. In Kim Larsen, Oleg Sokolsky, and Ji Wang, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10606 LNCS, pages 319–336. Springer, 2017. ISBN 9783319694825. doi: 10.1007/978-3-319-69483-2{_}19. URL http://link.springer.com/10.1007/978-3-319-69483-2_19. 3, 20, 24

Bruce Schneier. Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999. 22, 31

Daniel Sinnig, Maik Wurdel, Peter Forbrig, Patrice Chalin, and Ferhat Khendek. Practical Extensions for Task Models. In *Task Models and Diagrams for User Interface Design*, volume 4849 LNCS, pages 42–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 9783540772217. doi: 10.1007/978-3-540-77222-4_5. URL http://link.springer.com/10.1007/978-3-540-77222-4_5. 8

Daniel Sinnig, Ferhat Khendek, and Patrice Chalin. Partial order semantics for use case and task models. *Formal Aspects of Computing*, 23(3):307–332, may 2011. ISSN 0934-5043. doi: 10.1007/s00165-010-0158-z. URL http://link.springer.com/10.1007/s00165-010-0158-z. 22

Andreas Wolff, Peter Forbrig, Anke Dittmar, and Daniel Reichart. Linking GUI elements to tasks. In *Proceedings of the 4th international workshop on Task models and diagrams - TAMODIA '05*, volume 127, pages 27–34, New York, New York, USA, 2005. ACM Press. ISBN 1595932208. doi: 10.1145/1122935.1122941. 1