Translating Task Models to State Machines

Andreas Wagner[1] and Christian Prehofer[2]
[1]*itestra GmbH, München, Germany*[3]
[2]*fortiss GmbH, München, Germany*

Keywords: Task Models, Statecharts, Partial State Machines, Model Transformation, Interaction Modeling.

Abstract: We present a new method to translate the established and well-known ConcurTaskTree (CTT) task modeling technique into state machines. For this purpose, we develop the concepts of partial state machines, Connectables and a connect operator, which form the theoretical framework for a new algorithm. For the translation, we develop and present a recursive, bottom-up algorithm, which exploits the inherent structure of CTTs and produces valid, partial state machines. This allows new development processes in the model-driven application and system development domain.

1 INTRODUCTION

Task models are a widely used technique to model user interaction with a system. They describe a set of possible arrangements of basic tasks, which can be executed to achieve the overall goal. *ConcurTaskTrees (CTT)* (Paternò, 2000) are one of the most well-known approaches to task modeling. Key features of CTTs include focus on activities, hierarchical structure, graphical syntax and a rich set of temporal operators (Paternò, 2001). The hierarchical structure of CTTs allows for the decomposition of complex tasks into more fine-grained child tasks. Relationships between tasks are defined via temporal operators between adjacent siblings. For the purpose of this work, we focus on the key CTT operators *Enabling* (>>) , *Choice* ([]), *Concurrent* (|||) and *Disabling* ([>).

As an example, consider the (simplified) CTT of operating a TV as depicted in Figure 1. It consists of ten tasks, which implement the basic features of a TV like switching channels, adjusting the volume and putting the TV in standby mode. For instance, if the next channel should be selected, the user has to initiate the channel switch by pushing the "Chan+" button and the TV performs the necessary action. Further, channel switching and adjusting the volume should be possible to be done in parallel. By turning the TV off, all other tasks are interrupted.

In this work, we present a recursive algorithm to translate CTTs, based on their tree structure, into state machines. Contrarily to existing work on CTT-to-state-machine-translation (e.g. (Luyten and Clerckx, 2003) or (Van Den Bergh and Coninx, 2007)), our algorithm is a fully recursive, structure- and semantic-preserving approach to create executable state machines from CTTs. As the major technical contribution of this paper, we provide ways to manage and build intermediate state machines, which are incomplete or partial state machines and are composed in a recursive way. We develop and present the concepts of partial state machines and partial state machine containers.

One reason why the translation into state machines is challenging is that the actual input events are in the leaves of the CTT, but the CTT semantics is presented recursively in a top down fashion. Hence, a recursive algorithm must carefully compose the partial state machines and ensure at any time that the correct set of possible input events is maintained. This issue can be demonstrated in our TV example. The choice between the subtree "Increase Volume" (i.e. "Press Vol+ Button") and "Decrease Volume" (i.e. "Press Vol- Button") must be passed along the whole tree hierarchie, because both tasks may ultimately be used to as first interaction. A fully-recursive algorithm must consider this behaviour. While existing work in this area often relies on some precalculated information (so called "Enabled Task Sets", e.g. (Luyten and Clerckx, 2003) or (Van Den Bergh and Coninx, 2007)), our algorithm maintains the correct input events on the fly.

[3]research carried out at Technische Universität München

Wagner, A. and Prehofer, C.
Translating Task Models to State Machines.
DOI: 10.5220/0005681702010208
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 201-208
ISBN: 978-989-758-168-7

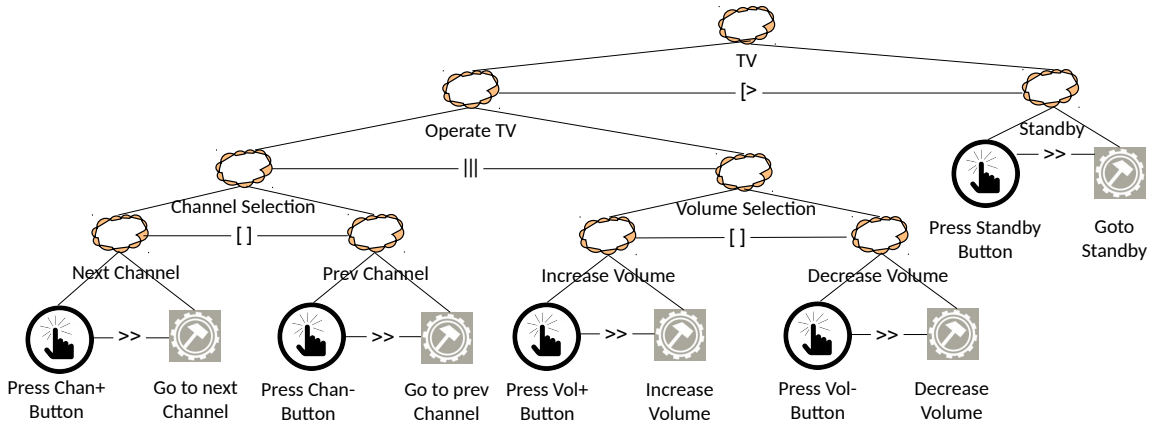

Figure 1: Simple CTT for operating a TV.

2 STATECHARTS

In this work, we follow statecharts as defined in (Eshuis, 2009) and (Pnueli and Shalev, 1991):

Definition 1. *A statechart SC is a tuple* $\tau = (S, T, E)$*, where*

- *S is a set of states*
- *T is a set of transitions*
- *E is the set of events that transitions*

For a given set S, which represents the set of all states and $\mathcal{P}(S)$ being the powerset of S, the function $substates\colon S \mapsto \mathcal{P}(S)$ maps a given state s to the set of its immediate children.

States may have a property *initial*, which indicates if a given state is an initial state. The function $isInitial\colon S \mapsto \{true, false\}$ assigns each state an indicator whether it is initial or not. **Basic states** are defined by means of the *substates* function. If the substates set is empty for a state $s \in S$, i.e. $substates(s) = \emptyset$, then s is called a "basic state". Basic states may have a *final* property (Omg, 2004). To check whether a basic state is final or not, we use the function $isFinal\colon S_B \mapsto \{true, false\}$, which for an element of the set of all basic states S_B indicates if the state is final or not.

Like basic states, **composite states** are defined by means of the *substates* function. If the substates set is not empty for a state $s \in S$, i.e. $substates(s) \neq \emptyset$, then s is called a "composite state". Composite states must be either of type *AND* or *OR*. If S_C is the set of all composite states, the function $type\colon S_C \mapsto \{AND, OR\}$ assigns for each composite state its type (Pnueli and Shalev, 1991).

OR-States indicate sequential behavior (Eshuis, 2009). An OR-State is defined by means of the *substates* function and the *type* function (Pnueli and Shalev, 1991). If a state $s \in S$ is of type *OR* and has substates, i.e. $substates(s) \neq \emptyset \wedge type(s) = OR$ it is called "OR-State". The set of all OR-States is denoted S_{OR}. It especially holds that when the statechart is in state s, it is also in one *and only one* state $s' \in substates(s)$.

Throughout this work, we call OR-States **compound** or **hierarchical** states. Each compound state s has an initial state s_{init}, which is an element of $substates(s)$. The initial state can be obtained by the function $initial\colon S_{OR} \mapsto S$.

AND or parallel states indicate parallel behavior (Eshuis, 2009). Analogously to OR-States, AND-States are defined by means of the *substates* and *type* function (Pnueli and Shalev, 1991). If a state $s \in S$ is of type *AND* and has substates, i.e. $substates(s) \neq \emptyset \wedge type(s) = AND$ it is called "AND-State".

When the statechart is in state s, it is simultaneously in all states $s' \in substates(s)$. Further, we require the substates $s' \in substates(s)$ to be composite states.

Like states, transitions have properties which can be obtained by corresponding functions. First, we denote the set of all transitions T and introduce the functions *source* and *target* on it. Both $source\colon T \mapsto S \cup \{null\}$ and $target\colon T \mapsto S \cup \{null\}$ are partial functions which for a transition $t \in T$ either return a $s \in S$ or *null* if $source(t)$ respectively $target(t)$ is undefined.

More details on the employed functions and predicates for state machines can be found in (Wagner, 2015).

3 PARTIAL STATE MACHINES

Partial State Machines will be be used as intermediate results of the translation algorithm. To combine par-

tial state machines, we use the concept of Connectables and a connect operator, as defined below.

Connectables are basically well-defined interfaces for the composition of partial state machines. Connectables rely on **free** transitions, which we define first. A transition $t \in T$ is said to be free iff $target(t) = null \vee source(t) = null$. We denote the set of all free transitions T_{free} and the complementary set of non-free transitions $T \setminus T_{free} = T_{nonfree}$.

With the definition of free transitions at hand, Connectables can be defined:

Definition 2. *Let S_B be the set of all basic states, S_C be the set of all compound states and S_P be the set of all parallel states. Further, let T_{free} be the set of all free transitions. A Connectable c is then defined as*

$$c \in (S_B \cup S_C \cup S_P \cup T_{free})$$

Consequently, a Connectable is an abstraction over states and transitions. Often the actual type of a Connectable is needed. For this purpose, we use the function $role\colon C \mapsto R$, where $R = \{\text{Transition}, \text{BasicState}, \text{CompoundState}, \text{ParallelState}\}$ is the set of all roles and C is the set of all Connectables.

Partial state machines (PSMs) are a superset of statecharts, which includes "incomplete" statecharts where e.g. transitions may not lead to a state. We define PSMs as follows:

Definition 3. *A partial state machine is a tuple $\tau = (S, T, E, \delta)$ where*

- *$S \subset (S_B \cup S_C \cup S_P)$ is a set of states*
- *$T \subset (T_{free} \cup T_{nonfree})$ is a set of transitions*
- *E is a set of events that trigger transitions*
- *$\delta\colon S\ x\ T \mapsto S$ is a partial transition function*

S and T may be empty, however the restriction holds that if $S = \emptyset$, $T \neq \emptyset$ and vice versa.

Figure 2 shows - in bold lines - some examples for (simple) partial state machines (the dotted lines are explained below).

Next we introduce *in* and *out* sets on PSMs. Both sets are in essence a set of the "dangling" components of the PSM. As *in* set I we denote all free transitions of a PSM which don't have a source state and all states which don't have ingoing transitions. Consequently, the *out* set O of a PSM consists of all free transitions which don't have a target state and all states which don't have outgoing transitions. Note that both I and O may contain basic, compound or parallel states. Further, we define that I consists of only transitions or exactly one state. As an example consider the PSM in Figure 2(d). The *in* set consists of the two (free) transitions "EventA_x" and "EventA_y", whereas the *out* set consists of the two (free) transitions "EventB_x" and "EventB_y". A formal definition of *in* and *out* sets can be found in (Wagner, 2015).

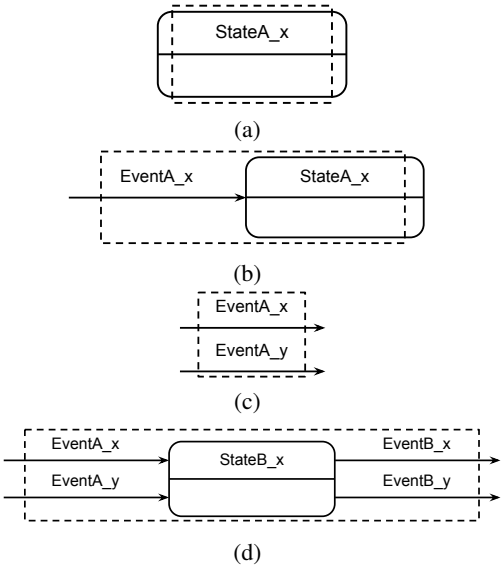

Figure 2: Examples for partial state machines and corresponding containers (in dotted lines).

Finally, we need a function $states\colon P \mapsto \mathcal{P}(S)$, which returns all the states (basic states, compound states, parallel states) of a given PSM. With the *in* sets, *states* set and *out* sets, we can define the concept of PSM containers:

Definition 4. *Let P be a partial state machine, T_P the set of transitions of P and S_P the set of states of P. A partial state machine container P_C is then a 3-tuple $\tau = (I_P, O_P, S_P)$ where*

- *I_P is the in set of P with $I_P \neq \emptyset$*
- *O_P is the out set of P with $O_P \neq \emptyset$*
- *S_P is the set of states of P*

Obviously, a PSM container is an abstratcion over a PSM, which hides the internal structure and only exposes its interfaces (i.e. Connectables) to the outside world. Figure 2 shows - in dotted lines - the PSM containers for the PSMs in this figure. Note that the corresponding container of a PSM can be calculated at runtime by means of the *in*, *out* and *states* functions. Therefore, we will often use the terms partial state machine (*PSM*) and partial state machine container (PSM_C) interchangeably throughout this work. This is only for convenience and does not have any implications on the translation algorithms.

Finally, the *connect*-operator (denoted by $\xrightarrow{connect}$) is a binary function which maps Connectables to sets of states and transitions. It takes two Connectables c_1 and c_2 as parameters and returns a set consisting of states and transitions with modified properties.

Most importantly, the $\overset{connect}{\longrightarrow}$-operator guarantees that the Connectables c_1 and c_2 are connected in a proper way, i.e. a state is followed by a transition and vice versa. Informally, its purpose is to check if two Connectables can be connected and to return the possible connections. We will use it to connect two Connectables of different partial state machines.

The $\overset{connect}{\longrightarrow}$-operator is formally defined as follows:

Definition 5. *Let C be the set of all Connectables, $S = S_B \cup S_C \cup S_P$ be the set of all states and $T = T_{free} \cup T_{nonfree}$ be the set of all transitions. Further, let $X = S \cup T$ and $\mathcal{P}(X)$ the powerset of X. Then the function $\overset{connect}{\longrightarrow}$ is defined as*

$$\overset{connect}{\longrightarrow} : C \times C \mapsto \mathcal{P}(X)$$

For its behavior, four cases have to be distinguished. If a Connectable c_1 is a (free) transition and a Connectable c_2 is a (free) transition, then c_1 is connected to c_2 by means of an intermediate basic state $s_{intermediate}$. The *target* of c_1 then points to $s_{intermediate}$, the *outgoingTransitions* set of $s_{intermediate}$ contains c_2, the *ingoingTransitions* set of $s_{intermediate}$ contains c_1 and the *source* of c_2 points to $s_{intermediate}$.

If c_1 is a (free) transition and c_2 is a state, then c_1 is directly connected to c_2. The *target* of c_1 then points to c_2 and the *ingoingTransitions* set of c_2 contains c_1. The reverse case is handled analogously. Finally, if c_1 is a state and c_2 is a state, then c_1 is connected to c_2 by means of an intermediate transition $t_{intermediate}$. The *outgoingTransitions* set of c_1 and the *ingoingTransitions* set of c_2 then contain $t_{intermediate}$, the *target* of $t_{intermediate}$ points to c_2 and the *source* of $t_{intermediate}$ points to c_1.

4 TRANSLATION RULES

The translation of CTTs into state machines is based on a recursive algorithm, where each recursive call returns a partial state machine. The algorithm is described in full detail (including proof-sketches employing induction) in (Wagner, 2015).

We start by explaining the basic case of the recursion, namely tasks. Tasks are the leafs of a CTT. Within an application, they perform the actual computations, enable the user to interact with the system, or provide feedback to the user and the environment. For the translation, we only cover **interaction tasks** and **application tasks**. The type of a task directly influences its counterpart in the resulting PSM.

Interaction tasks require some sort of interaction with the user and/or environment, thus being always bound to a specific event. Events move the system forward and may initiate an arbitrary computation or response from the system. In state machines, events are represented as transitions (e.g. (Harel, 1987)). Thus, we translate an interaction task into a **transition**.

On the other hand, it is statically known that an application task is performed on the device and does not require any interaction with the user. Because of this, an application task can be seen as an isolated computation unit, which is entered when the computation must be done and left when the computation finished. We therefore translate an application task into a **basic state**.

Besides the events which correspond to interaction tasks, additional events must be introduced which notify the system about the completion of application tasks (i.e. that states finished their execution). These events are called notification events and trigger *notification transitions*, which are attached to the states which correspond to the application tasks.

As an additional step, the generated states and transitions must be wrapped into a partial state machine container $PSM_{Interaction}$ and $PSM_{Application}$ respectively, which provide the Connectables for subsequent connection operations. For interaction tasks, the transition is the only element created, it is simply returned as both *in* and *out* set of the PSM. For application tasks, the created state is returned as *in* set and the notification transition is returned as *out* set.

With the translation rules for the base case (i.e. tasks) at hand, we can now translate the CTT operators. In the follwing, we consider translations for the *Enabling*, *Choice*, *Concurrent* and *Disabling* operators. As other operators do not add essentially new concepts, we expect that the translation scheme can be extended to other operators analogously.

Note that CTT execution is in general nondeterministic, as there are choices to be taken in *Choice*, *Concurrent* and *Disabling* operators. We will assume that these choices are essentially taken by external user interactions and not by random decisions. In case such non-determinism is desired, it can be modeled by external components which take the required decisions. For the purpose of specifying our assumption, we use the function *first*, which is introduced in (Paternò, 2000). The informal purpose of this function is to return a set of tasks which should be executed first, given a CTT or a subtree. We require that the $first$ set for all *Disabling*, *Concurrent* and *Choice* operators consists of only interaction tasks (i.e. tasks which require user interaction). Thus we can make sure that choices are always resolved externally (i.e. by the user).

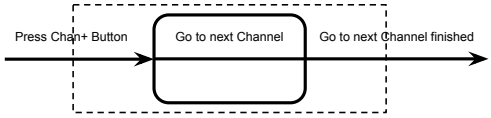

Figure 3: $PSM_{Enabling}$ for sub-CTT "Next Channel".

4.1 Enabling Operator

According to the CTT specification provided in (Paternò et al., 2012), the semantics of *Enabling* requires the second task not to be activated until the first task is performed. This establishes a strictly sequential relation between the tasks of the operator. The partial state machine representing the *Enabling* operator must guarantee that this relation between α and β is preserved during the execution phase (i.e. that α *must* be executed before β in any case).

The actual translation of a CTT $\alpha >> \beta$ is shown in Algorithm 1. To seize on our example of Figure 1, we apply the algorithm to the *Enabling* operator with α being the task "Press Chan+ Button" and β being the task "Go to next Channel". First, the algorithm creates partial state machine containers PSM_α for α and PSM_β for β by applying the recursive translation function $translate$ to both tasks (or subtrees in general). The connection between both PSMs is done by connecting each component of $out(PSM_\alpha)$ (which is a transition representing the "press button" task) with each component of $in(PSM_\beta)$ (a state representing the "goto" task) by means of the $\overset{connect}{\longrightarrow}$-operator. Note that in general, $in(PSM_\beta)$ either contains only transitions or exactly one state. This is ensured by the definition of the *in* set and the aforementioned requirements on the CTT.

The result of the connection operation is a new partial state machine $PSM_{Enabling}$ that contains transitions and states from PSM_α and PSM_β as well as additional states introduced by the $\overset{connect}{\longrightarrow}$-operator (Figure 3). The *in* set of $PSM_{Enabling}$ is the *in* set of PSM_α and the *out* set of $PSM_{Enabling}$ is the *out* set of PSM_β. Note that for further connections, only the Connectables of $PSM_{Enabling}$ are relevant. The contents of the PSM container do not affect the translation of the upper operators.

4.2 Choice Operator

The *Choice* operator is described by Paternò to offer two tasks/subtrees and "once one has started the other one is no longer enabled" (Paternò et al., 2012). For our translation algorithm, we assume that choices always start with interaction tasks, as defined beforehand. Thus, we avoid any kind of of possible non-determinism, because the choices are "externally" resolved by the user via explicit actions (e.g. input events).

Algorithm 1: $translate(Enabling(\alpha,\beta))$.

$PSM_\alpha \leftarrow$ translate(α)
$PSM_\beta \leftarrow$ translate(β)
inter $\leftarrow \emptyset$
for all $o \in out(PSM_\alpha)$ **do**
  for all $i \in in(PSM_\beta)$ **do**
    $o \overset{connect}{\longrightarrow} i$
    if new state was created **then**
      inter $\leftarrow$ inter $\cup$ {state}
    end if
  end for
end for
return PSM container $PSM_{Enabling}$

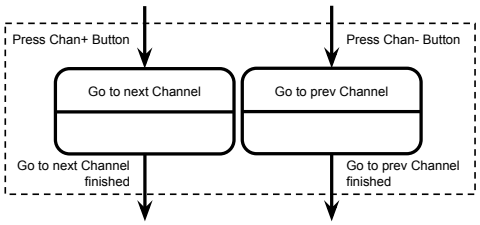

Figure 4: PSM_{Choice} for sub-CTT "Channel Selection". Because we require *Choice* operators to start with interaction tasks (and therefore transitions/events) only, the user can decide which subtree should be executed.

The idea is to generate two, mutually exclusive PSMs, where each PSM can be entered via its own transition(s). The *in* set of both state machines thus represents the "decision", which subtree should be executed. Once PSM_α is active, there is no possibility to enter PSM_β and vice versa.

The translation rule is shown by Algorithm 2. Considering our example, we translate the *Choice* between going to the next channel (α) or to the previous channel (β). The main step is to generate partial state machines PSM_α and PSM_β by applying the translation function $translate$ to both subtrees.

After translating the subtrees, we can merge both PSMs into a combined partial state machine PSM_{Choice}, which leads to a single PSM container representing the operator (Figure 4).

Algorithm 2: $translate(Choice(\alpha,\beta))$.

$PSM_\alpha \leftarrow$ translate(α)
$PSM_\beta \leftarrow$ translate(β)
return PSM container PSM_{Choice}

4.3 Concurrent Operator

The *Concurrent* operator allows independent and concurrent execution of both subtrees α and β. This

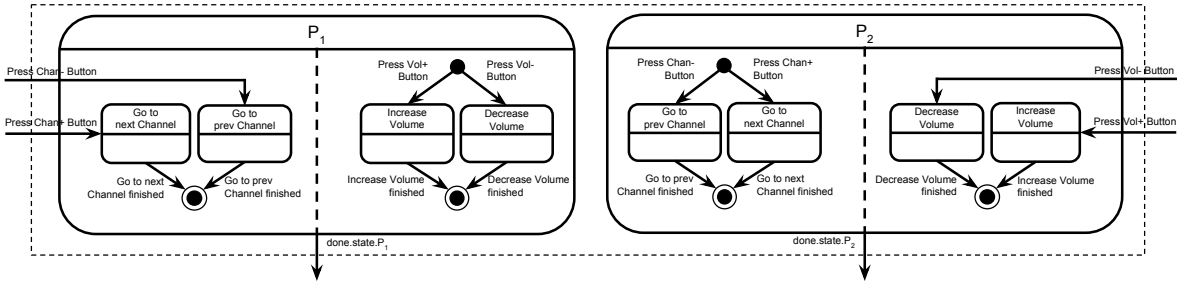

Figure 5: $PSM_{Concurrent}$ for sub-CTT "Operate TV". The translation rule creates two parallel states in order to avoid that the *in* set of the PSM becomes mutually exclusive.

means, that at every step in time a task of α or β can be executed without effecting each other. Paternò et al. define the tasks of the *Concurrent* operator to be "performed in any order, or at same time, including the possibility of starting a task before the other one has been completed" (Paternò et al., 2012). Consequently, all interleavings of the tasks of α and β must be possible.

It is therefore feasible to translate the *Concurrent* operator directly into parallel states. Each subtree is represented by a substate of a common parallel state, which allows the corresponding tasks to be executed independently. The algorithm basically creates two parallel states - one with Connectables (transitions) for α and one with Connectables (transitions) for β. This is necessary in order to preserve the semantics of the *Concurrent* operator. If only one parallel state with both Connectables for α and β would be created, the ingoing transitions would be mutually exclusive, which leads to a violation of the operator's semantics.

The formal translation steps for *Concurrent* are shown by Algorithm 3. In our example, the sub-CTT "Operate TV" matches the translation rule with α being the subtree "Channel Selection" and β being the subtree "Volume Selection". For the translation, we first create corresponding partial state machines PSM_α and PSM_β by applying *translate* to both subtrees. Next, PSM_α will be equipped with a new final state by connecting each transition of PSM_α's *out* set to it. This leads to a new partial state machine $PSM_{\alpha_{final}}$. PSM_β ("Volume Selection") on the other hand is attached with a new initial and final state. This turns the partial state machine into an ordinary executable state machine which we call $PSM_{\beta_{exec}}$. Note that $in(PSM_\beta) \neq in(PSM_{\beta_{exec}})$, because the transitions of PSM_β's *in* set were replaced by the common initial state.

Next, the ingoing transitions $transitions_\alpha$ of $PSM_{\alpha_{final}}$ are extracted and both $PSM_{\alpha_{final}}$ and $PSM_{\beta_{exec}}$ are wrapped into their own compound state C_1^α and C_1^β. Note that the ingoing transitions "Press Chan+ Button" and "Press Chan- Button" of $PSM_{\alpha_{final}}$ still point to their targets in C_1^α. C_1^α and C_1^β are then wrapped in a parallel state P_1. Finally, a completion transition $t_1^{completion}$ is created which is connected with P_1. The completion transition will be triggered automatically, if all of the parallel state's substates reach their final state. The parallel state with Connectables for β is created analogously, but with PSM_α and PSM_β interchanged.

$PSM_{Concurrent}$ of the CTT "Operate TV" is shown in Figure 5.

Algorithm 3: $translate(Concurrent(\alpha,\beta))$.

$PSM_\alpha \leftarrow$ translate(α)
$PSM_\beta \leftarrow$ translate(β)
$PSM_{\alpha_{final}} \leftarrow$ CreateFinalStates(PSM_α)
$PSM_{\beta_{exec}} \leftarrow$ CreateExecutionClosure(PSM_β)
$transitions_\alpha \leftarrow$ all transitions from $in(PSM_{\alpha_{final}})$
$C_1^\alpha \leftarrow$ CompoundState($cname_1$,states($PSM_{\alpha_{final}}$))
$C_1^\beta \leftarrow$ CompoundState($cname_2$,states($PSM_{\beta_{exec}}$))
$P_1 \leftarrow$ ParallelState($pname_1$, $\{C_1^\alpha, C_1^\beta\}$)
$t_1^{completion} \leftarrow$ Transition("done.state.$pname_1$")
$P_1 \xrightarrow{connect} t_1^{completion}$
repeat with α and β interchanged
return PSM container $PSM_{Concurrent}$

4.4 Disabling Operator

The *Disabling* operator has two subtrees α and β, where α is regularly executed but can be "completely interrupted" (Paternò et al., 2012) by β at any time. The remaining tasks of α are not executed once β becomes active. This also implies that α is never executed if (one of) the first task(s) of β is selected for execution before the first task(s) of α.

From the *Disabling* operator point of view, α can be regarded as a compound state containing a partial state machine PSM_α. A partial state machine PSM_β (more precise, the *in* set of PSM_β) which implements β is an "interrupting" partial state machine, which exits (or bypasses) the compound state containing PSM_α

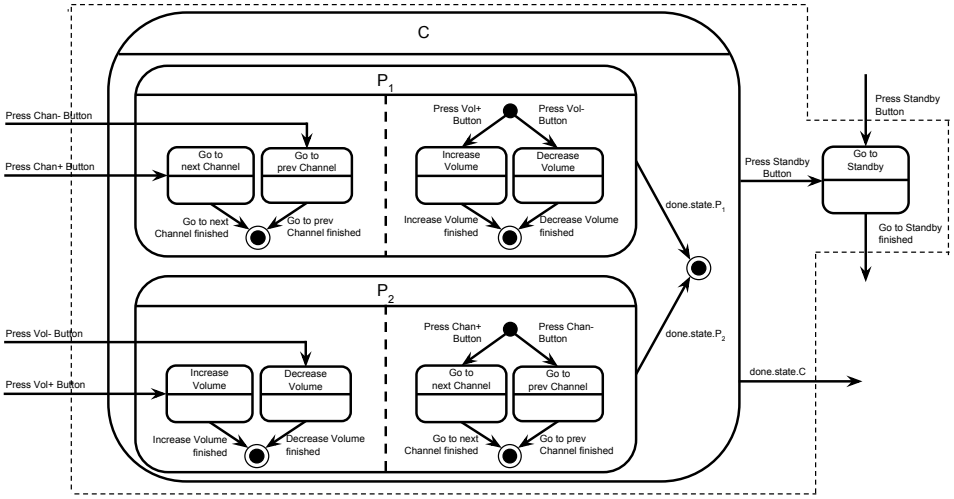

Figure 6: $PSM_{Disabling}$ of the CTT "TV". Note that the *in* set contains both ingoing transitions of the compound state and transitions of PSM_β to bypass the execution.

and switches to another execution branch.
To finish the execution of α without interruption, the compound state representing α is equipped with an internal *final* state, which is entered when α (i.e. PSM_α) was successfully executed. Due to the corresponding completion transition, the compound state is automatically left when α (PSM_α respectively) was executed without interruption of β (PSM_β respectively).

The formal translation steps of the *Disabling* operator are shown by Algorithm 4. Again, we apply the algorithm to the example of Figure 1. Because *Disabling* is the top-most operator, applying the algorithm completely translates our CTT. α in this case is the subtree "Operate TV" and β is the subtree "Standby". As usual, PSM_α and PSM_β are generated by applying *translate* to the corresponding sub-CTTs. Next, we create the compound state for PSM_α, which requires an additional final state that is connected to the *out* set (i.e. the transitions $t_1^{completion}$ and $t_2^{completion}$) of PSM_α. This results in a new partial state machine $PSM_{\alpha_{final}}$.

Before wrapping $PSM_{\alpha_{final}}$ into the compound state, the ingoing transitions "Press Chan+ Button", "Press Chan- Button", "Press Vol+ Button" and "Press Vol- Button" of $in(PSM_{\alpha_{final}})$ must be preserved for subsequent connections. Consequently, the transitions of both $in(PSM_{\alpha_{final}})$ and $in(PSM_\beta)$ are extracted and the transitions of $in(PSM_\beta)$ ("Press Standby Button") are duplicated to provide Connectables that can bypass PSM_α and its corresponding compound state completely.

As *in* set of the resulting container $PSM_{Disabling}$, the extracted transitions of PSM_α and the duplicated ingoing transitions of PSM_β are returned. The *out* set consists of the completion transition $t_{completion}$ and the *out* set of PSM_β. Consequently, the translated *Disabling* operator of the TV example is shown in Figure 6. Note that the transitions belonging to PSM_α point **into** the compound state and not just to it.

With the root operator of the CTT translated, we obtained a complete (yet partial) state machine from the CTT. However, this PSM is not executable yet, because it lacks proper initial and final states. By adding a common initial state, which is connected to the transitions of the $PSM_{Disabling}$'s *in* set and a common final state, which is connected with $PSM_{Disabling}$'s *out* set, we obtain a state machine which can be executed by standard state machine frameworks. We call this final step the creation of the "execution closure".

Algorithm 4: $translate(Disabling(\alpha,\beta))$.

$PSM_\alpha \leftarrow$ translate(α)
$PSM_\beta \leftarrow$ translate(α)
$PSM_{\alpha_{final}} \leftarrow$ CreateFinalStates(PSM_α)
$transitions_\alpha \leftarrow$ all transitions from $in(PSM_{\alpha_{final}})$
$transitions_\beta \leftarrow$ all transitions from $in(PSM_\beta)$
$transitions_\beta^{dup} \leftarrow$ copy of $transitions_\beta$ with target
C $\leftarrow$ CompoundState(*cname*, states($PSM_{\alpha_{final}}$))
intermediate $\leftarrow \emptyset$
for all $i \in in(PSM_\beta)$ **do**
  C $\xrightarrow{connect}$ i
  if state was created **then**
    intermediate $\leftarrow$ intermediate $\cup$ {state}
  end if
end for
$t_{completion} \leftarrow$ Transition("done.state.*cname*")
C $\xrightarrow{connect}$ $t_{completion}$
intermediate $\leftarrow$ intermediate $\cup$ $\{t_{completion}\}$
return PSM container $PSM_{Disabling}$

5 RELATED WORK

Task models have been found to be widely adapted in the research community. Several related work covers the translation of task models into corresponding UML diagrams, but in a limited way. For example, (Luyten and Clerckx, 2003; Van Den Bergh and Coninx, 2007) define mappings from CTTs to state machines, but neither provide a formal set of rules nor do they cover the full recursive case. Other work maps CTTs to UML activity diagrams in a similar way as in this work (Brüning et al., 2008; Brüning et al., 2012). However, we feel that state charts are easier to handle and use in actual implementations. Furthermore, (Zhu et al., 2012) extract state machines from CTTs to derive the system behavior. Unfortunately, they only use it for verification, but do not consider interactive execution. A recursive algorithm to map CTTs to state machines is presented in (Sinnig et al., 2013), but this only aims at extracting possible executions for verifications, and also does not consider interactive execution. Compared to this work, related projects that employ (CTT) task models to derive system information mostly seem to concentrate on the user interface (which is not enough when focusing a more comprehensive approach to model-driven systems engineering) or they generate UML models on a higher and more abstract level, which require some manual effort to implement.

6 CONCLUSION

In this work, we have presented the first recursive algorithm to translate CTTs, based on their tree structure, into executable state machines. Compared to prior work, the generated state machines interact with the environment by means of high-level events which correspond to tasks in CTTs. These events can be triggered by actual UI elements or concrete (sensor) events in real implementations. A full tool chain for this is presented in (Wagner, 2015). As the major technical contribution of this paper, we have introduced and extensively used incomplete or partial state machines. These are composed in a recursive way to form bigger and more complex (partial) state machines, which can act as central controllers for desktop or embedded applications.

REFERENCES

Brüning, J., Dittmar, A., Forbrig, P., and Reichart, D. (2008). Getting SW engineers on board: Task modelling with activity diagrams. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4940 LNCS:175–192.

Brüning, J., Kunert, M., and Lantow, B. (2012). Modeling and executing concurtasktrees using a uml and soil-based metamodel. In *Proceedings of the 12th Workshop on OCL and Textual Modelling*, OCL '12, pages 43–48, New York, NY, USA. ACM.

Eshuis, R. (2009). Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65–99.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Luyten, K. and Clerckx, T. (2003). Derivation of a dialog model from a task model by activity chain extraction. *Interactive Systems. Design . . .* , pages 203–217.

Omg (2004). UML 2.4.1 Superstructure Specification. *October*, 02(August):1–786.

Paternò, F. (2000). *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag London.

Paternò, F. (2001). Task models in interactive software systems. *Handbook of software engineering & knowledge engineering*, pages 817–836.

Paternò, F., Santoro, C., and Spano, L. D. (2012). Concur Task Trees (CTT).

Pnueli, A. and Shalev, M. (1991). What is in a step: On the semantics of statecharts. 937:244–264.

Sinnig, D., Chalin, P., and Khendek, F. (2013). Use case and task models: an integrated development methodology and its formal foundation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):27.

Van Den Bergh, J. and Coninx, K. (2007). From Task to Dialog Model in the UML. In *Task Models and Diagrams for User Interface Design*, pages 98 – 111.

Wagner, A. (2015). Multi-Device Extensions for CTT Diagrams and their Use in a Model-based Tool Chain for the Internet of Things. Master's thesis, TU München, Germany.

Zhu, B., Zhang, S., and Wang, A. (2012). Towards a formal integrated model for function and user interface. *Proceedings - 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, SNPD 2012*, pages 275–280.