A Model-based Approach for Multi-Device User Interactions

Christian Prehofer
Fortiss GmbH
München, Germany
prehofer@fortiss.org

Andreas Wagner*
Technische Universität München
München, Germany
andreas.2.wagner@tum.de

Yucheng Jin†
Fortiss GmbH
München, Germany
jiny@in.tum.de

ABSTRACT

This paper presents an approach for modeling multi-device user interactions, based on task models. We use ConcurTaskTrees (CTTs) as a domain-specific language, which we extend here by a labeling mechanism to model multi-device interactive applications. While CTTs are used to specify temporal and causal relations between tasks, we add operators to specify the device mapping in a flexible and expressive way. The main novelty is the introduction of the two new operators, *Any* and *All*, to specify if a task should be executed on any or on all of a set of devices. We show that this is applicable in scenarios of connected, smart devices where a task can be executed on a multitude of devices. We present formal semantics for our extension of CTTs as well as a tool chain based on the Qt toolkit for generating code for distributed UIs. This includes a mapping from high-level tasks to concrete UI controls and a distributed execution model based on state machines. The new concepts are validated in several case studies.

Keywords

Task models, multi-device UI, cross-device UI, Model-based Development and Tools, Multi-device Applications

1. INTRODUCTION

This paper presents an approach for modeling multi-device user interactions. For several years we have observed a paradigm shift in device and software usage scenarios. Instead of just having a single device for performing input and output activities for an application, we see that users tend to interact with a system via multiple devices. For instance, users may control a TV via a remote control, a mobile device like a smartphone or via the device itself. Hence, there is considerable interest in multi- or cross-device interfaces, as indicated in a recent workshop on the topic [18]. Several researchers have considered such "multi-device applications", which connect multiple devices to work collaboratively at the same time, e.g. [3, 27].

In the context of user interaction modeling, ConcurTaskTrees (CTT) are a common technique to model the flow of activities performed to reach a goal on an abstract level [20]. Task models can be seen as domain specific languages [25], which describe relations between tasks in a hierarchical way, e.g. sequences of tasks, tasks disabling other tasks or concurrent execution. Such task models can be translated into a more verbose representation with parallel and hierarchical state machines, as shown in [31]. In order to cope with different devices, several approaches exist to specify the deployment of user tasks to devices [24], mainly by simple annotations on the models.

In this paper, we cover the case where the assignment of tasks to devices occurs *dynamically* during runtime, which is not possible with current CTT-based approaches. This assumes that a user task can be executed on one of several devices, or possibly on multiple devices. As an example, consider a connected microwave which can be controlled from the usual panel with buttons on the device, but also via a mobile device using an application. In this example, some operations can be executed on any of the devices, e.g. starting the microwave, while others must be executed on a specific device, e.g. opening the door. Further, it could be desirable to execute some tasks on all devices, e.g. showing the operational status of the microwave or raising some alarms.

In addition, there can be *dependencies* in the assignment of devices to tasks, e.g. starting an operation on a device implies that subsequent tasks must also be executed on this device. For instance, when setting the cooking timer, we would not expect that the minutes are set on the mobile and the seconds on the microwave itself. On the other hand, if we configure a more complex cooking procedure on one specific device, we might still want to express that the user can cancel the cooking process on some other device at any time. This coordination of tasks on multiple devices is mentioned as a main issue for cross-device user interfaces in [21]. Here, we address this both on the modeling and also on the execution level by our tool chain and execution model.

To model such multi-device user interactions, we introduce a multi-device extension for CTTs, which we call MCTT (Multi-device CTT). Contrarily to the existing CTT mechanism, these multi-device extensions not only specify how

*Current affiliation: itestra GmbH, Germany, wagner@itestra.de
†Current affiliation: KU Leuven, Belgium, yucheng.jin@cs.kuleuven.be.

MODELS '16, October 02 - 07, 2016, Saint-Malo, France

ISBN 978-1-4503-4321-3/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2976767.2976776

tasks can be *mapped* to devices, but also express the *relations* of the involved devices in the context of a given task or task configuration. Thus, the new multi-device extensions add a new spatio-temporal dimension to the CTT domain, while the original CTT operators express the relations between tasks.

In more detail, we introduce CTT-like operators and their corresponding semantics to specify that tasks can either be executed on one, several or all of the available devices. Compared to device mappings of tasks as described in existing work, we treat the new device operators as "first class citizens" like any other CTT operator. Thus, we can specify device mappings not just for single tasks, but for whole subtrees in CTTs. For instance, consider a cooking program on a microwave that includes several steps, which must be executed on exactly one of a set of devices. Using MCTTs, we can specify this by the operator $Any^{mi,mo}$ to state that the cooking program can be executed on either the microwave (mi) or the mobile device (mo). However, if a specific step in the CTT should be executed on the microwave exclusively, e.g. closing the microwave door, we can specify this by labeling the corresponding task with Any^{mi}. This can be seen as an exception to the specification of a device mapping on the whole tree and must be considered when executing the MCTT. In the existing literature, there is little work on such dependencies. Only the work in [5] covers the concept of delegation, which is a specific case of a dependency.

The focus of our approach lays on UI interaction tasks which can be executed on different devices. In general, the same abstract interaction task can be implemented differently on different devices, see e.g. [22]. Our goal is to specify the user interaction on a high level, and then map it to different UI elements, depending on the device capabilities. Consequently, we also present a tool chain based on the well-known Qt toolkit, which creates distributed state machines that are tailored to specific devices. It includes a mapping from high-level tasks to concrete UI controls and a distributed execution model based on distributed but synchronized state machines. Furthermore, this requires dynamic coordination of the tasks selected at runtime.

The remainder of this paper is as follows. In the next sections we introduce CTTs and the MCTT extension, including the semantics. The specifications have been implemented in a tool chain, including a distributed execution model, which is presented in Section 4. We also included a technique to map these generic, multi-device tasks to device-specific UI controls. An example of how to specify MCTTs and develop (simulated) devices which employ our techniques is provided in Section 5.

2. MULTI-DEVICE TASK MODELS

In order to introduce the concept of multi-device CTTs, we first recap task models and the CTT notation. Tasks are activities that have to be performed to reach a goal, which in turn can be described as a desired modification of the state of an application or an attempt to retrieve some information from an application [20]. Task models are usually used for design, evaluation and documentation purposes, but in the more recent past, they have also been employed to derive system specifications, particularly user interfaces [1, 29, 11].

The question of how to represent task models has lead to several proposals for task model notations. *ConcurTaskTrees* have emerged as one of the most widely used technique for this purpose.

Table 1: Overview of CTT operators and the MCTT extensions.

Operator	Symbol	Definition
$Ch(\alpha_1, \alpha_2)$	[]	**Choice:** One of the two choices is taken at run time.
$Co(\alpha_1, \alpha_2)$	|||	**Concurrent:** CTTs α_1 and α_2 are performed concurrently, with any interleaving of sub-tasks.
$Di(\alpha_1, \alpha_2)$	[>	**Disabling:** The CTT α_1 is executed and can be interrupted at any time by α_2, after which the execution continues in α_2.
$En(\alpha_1, \alpha_2)$	>>	**Enabling:** The CTT α_2 starts after the CTT α_1.
		MCTT Extension
$Any^{c_1,\ldots,c_i}(\alpha)$	$Any(\ldots)$	**Any:** The CTT α is executed on one of the devices $c_1, \ldots, c_i$.
$All^{c_1,\ldots,c_i}(\alpha)$	$All(\ldots)$	**All:** The CTT α is executed on all of the devices $c_1, \ldots, c_i$.

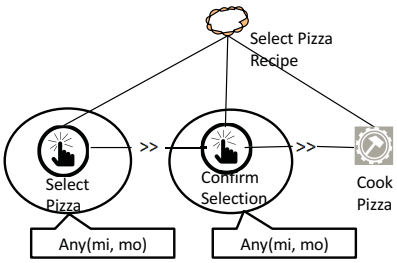


Figure 1: Example of labeling tasks

Models in CTT notation describe a set of possible sequences of basic tasks, which can be executed to achieve the overall goal [1]. The CTT notation defines four types of tasks [19]: *User tasks* are cognitive/perceptive and don't require interaction with the system. *Interaction tasks* involve user interaction, e.g. providing inputs or clicking a button. *Application or system tasks* are performed by the system without any additional user interaction. Finally, *abstract tasks* refer to complex tasks which have more concrete subtasks.

Table 1 shows the CTT operators used throughout this work from highest to lowest priority, including the textual notation as well as the symbols used in the graphical notation. For the purpose of this paper, we only use a subset of the CTT notation which consists of some basic CTT operators. We expect that the presented concepts can easily be extended to other operators.

2.1 Multi-device CTTs

Multi-device CTTs (MCTTs) extend the CTT notation in order to execute CTTs on multiple devices. Mainly, we introduce the two new operators *Any* and *All*, which are called device labeling operators and are used for specifying

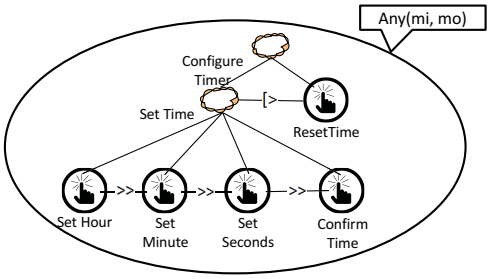


Figure 2: Example of labeling of subtrees

devices within a MCTT. These operators primarily attach a list of device identifiers to a CTT, a subtree within a CTT or a single task. For graphical representation, we use circles on the CTTs to denote the labeling operators. As depicted in Figure 1, we annotate the circles with $Any(mi, mo)$, which denotes that the labeled sub-CTT is to be executed on any of the devices. As introduced, *mi* stands for microwave and *mo* for mobile device. This notation easily maps to the operators shown in Table 1. Note that we use n-ary versions of the operators for convenience. Further, we use different symbols for different kinds of tasks. The finger icon indicates an interaction task (e.g. a user input), while the hammer icon indicates an application task (e.g. a system or output task).

The operator *Any* means that the tasks can only be performed on one of the specified devices, while the operator *All* specifies that the tasks are executed on all of the specified devices. The specified devices are indicated by the operator's attached device identifier list. In the example of Figure 1, the two interaction tasks "Select Pizza" and "Confirm Selection" can each be executed on any of the two specified devices. This means that the user can do the first action - the selection - on one device, and the second action - the confirmation - the other device. Thus, there may be a context switch (i.e. a switch of devices) between the two user interactions.

In case we wish to specify that a more complex task configuration takes place on any of the available devices (i.e. without such a switch of devices), we can label a whole subtree. Figure 2 shows a set of tasks for setting the current time on a microwave with the sub-tasks "Set Hour", "Set Minutes" and "Set Seconds" etc. The following *Disabling* indicates that the task "ResetTime" may interrupt the time setting at an arbitrary point during the setup.

By assigning an *Any* label to this task configuration, we enable the user to perform the tasks on any of the two devices, but once started on a device, it has to be completed on the selected device. This shows the expressiveness of our approach. It allows the specification of device mapping relations for whole sub-CTTs with corresponding execution semantics, which is not possible in prior approaches.

The *All* operator means that a CTT is to be executed on all of the specified devices. An example can be found in Figure 3, where an alarm can be configured on any of the two devices, but the alarm itself must take place on all devices. The *All* operator might typically be used for output actions (like alarms or notifications), but also other applications are possible when the same task has to be executed on multiple devices. A possible application might be some kind of barrier or synchronization point, which must be reached by all involved devices.

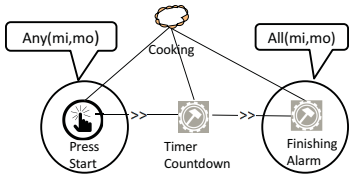


Figure 3: Execution of tasks with multiple labels

2.2 Exceptions in subtree labelings

Since *Any* and *All* can be applied to both tasks and whole subtrees, we can express more complex MCTTs by nesting these device operators. For instance, we might want to specify that all tasks in a CTT shall be performed on any of two devices, but some specific tasks in this CTT shall be executed on all devices.

Figure 4 shows such an example of cooking a pizza, which can be controlled either on the microwave or on the mobile. Yet the final notification must be shown on both devices. This *All* operator can be seen as an exception to the outer specification of *Any*.

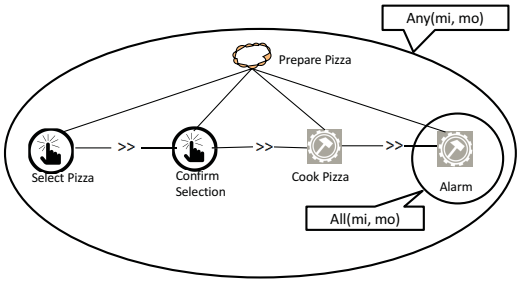


Figure 4: Exception of the labeling of subtrees

To illustrate the concept of device labels, we show a bigger task configuration integrating the above examples related to cooking on microwaves. In Figure 5, the MCTT model shows a use case of configuring a microwave. The model distinguishes between manual configuration and the selection of predefined recipes. For the latter case, we assume that selection of the pizza and the confirmation take place on one of the two devices. For the manual configuration option, several steps are needed, where the device can be selected individually for each interaction. Thus, it is possible to perform the tasks on one device, or change the device during this sequence.

3. OPERATIONAL SEMANTICS OF MCTTS

In the following, we introduce the behavior of our multi-device CTTs by semantical rules. The goal of device labeling is to retain the expressiveness and simplicity of the CTT notation, but also support the development of multi-device applications on a higher level of abstraction.

Mainly, we have to define semantics for the two new relational operators named *Any* and *All*. When multiple device

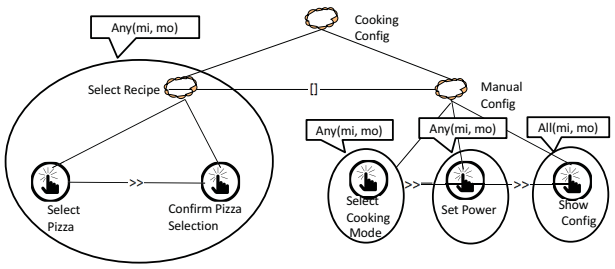


Figure 5: Example of MCTT for configuring cooking with a microwave

labels are assigned to a task or a set of tasks, *Any* means the tasks can only be performed on one of the defined devices at the same time. More precisely, one of the defined devices is selected at runtime for executing the task or task tree. *All* on the other hand enables the tasks to be executed on multiple devices.

To specify the semantics, we use the approach of a rewriting logic [15] which defines the operational behavior by rewrite rules. The major reason to use rewrite rules is to explicitly denote the task execution. Further, they allow us to easily apply simplification rules on the task tree. For instance, we will push the *All* and *Any* operators towards the leaves of the tree by means of rewrite rules.

We first define the semantics of well-known CTT operators and extend them with device labels which specify the devices that are associated to tasks. In the following, (sub) CTTs are denoted as α_1 and α_2 respectively and device labels are denoted c_i, where $i \in \mathbb{N}$. We also use a for atomic tasks (i.e. leaves). We write $emCTT$ for the empty CTT, which is typically the result of a complete execution.

We specify the operational semantics for MCTT device labels via execution steps of the following form:

$$\alpha_1 \xrightarrow{c(a)} \alpha_2$$

This denotes that a task a is executed on the device which is represented by the device label c. The semantics is defined for MCTTs where the outermost operator is *Any* or *All*, to ensure all tasks in the MCTT have a device mapping.[1]

Rewrite rules of the form

$$\alpha_1 \longrightarrow \alpha_2,$$

denote an internal simplification rule on CTTs without executing a task. We use this later to define *Any* and *All*. Further, we assume that rules forming an execution step with some task are applied in an outside-in fashion, i.e. starting at the top of the tree. Simplification rules may be applied in other places to facilitate the application of these execution steps.

A complete execution ends in the empty CTT by applying rewrite rules, e.g..

$$\alpha_1 \xrightarrow{c_1(a_1)} \ldots \longrightarrow \ldots \alpha_2 \xrightarrow{c_2(a_2)} \ldots \xrightarrow{c_k(a_k)} emCTT$$

As long as we do not have a looping or recursion operator in the CTTs, the executions will always terminate. We will add a repeat operator later for the examples.

[1]During execution, this may change as *All* and *Any* are pushed inside.

3.1 Semantics of CTT operators

As a prerequisite for our new device labeling operators, we specify the semantics of the conventional CTT operators first.

For a *Choice* operator we use the following rule:

$$Ch(\alpha_1, \alpha_2) \longrightarrow \alpha_i$$

for both $i = 1$ and $i = 2$. This means that different executions are possible for one CTT, each reflecting one possible behavior. Formally, the rules are not confluent.

For the *Enabling* operator we have a sequential execution of two CTTs:

$$En(\alpha_1, \alpha_2) \xrightarrow{c(a)} En(\alpha_1', \alpha_2) \text{ if } \alpha_1 \xrightarrow{c(a)} \alpha_1'$$
$$En(emCTT, \alpha_2) \longrightarrow \alpha_2$$

The *Concurrent* operator is defined by permitting an arbitrary interleaving of executions on the concurrent sub-CTTs:

$$Co(\alpha_1, \alpha_2) \xrightarrow{c(a)} Co(\alpha_1', \alpha_2) \text{ if } \alpha_1 \xrightarrow{c(a)} \alpha_1'$$
$$Co(\alpha_1, \alpha_2) \xrightarrow{c(a)} Co(\alpha_1, \alpha_2') \text{ if } \alpha_2 \xrightarrow{c(a)} \alpha_2'$$
$$Co(emCTT, emCTT) \longrightarrow emCTT$$

The *Disabling* operator executes the tasks of the first sub-CTT, until (one of) the first task(s) of the second CTT occurs. This leads to the following rule:

$$Di(\alpha_1, \alpha_2) \xrightarrow{c(a)} Di(\alpha_1', \alpha_2) \quad \text{if } \alpha_1 \xrightarrow{c(a)} \alpha_1'$$
$$Di(\alpha_1, \alpha_2) \xrightarrow{c(a)} \alpha_2' \quad \text{if } \alpha_2 \xrightarrow{c(a)} \alpha_2'$$
$$Di(emCTT, \beta) \longrightarrow \beta$$

3.2 Semantics of the device labeling operators

With the CTT operators defined as rewrite rules, we can present the semantics of the *Any* operator:

$$Any^{c_1,\ldots,c_i}(\alpha)$$

denotes that the CTT α shall be executed on exactly **one** of the devices $c_1, \ldots, c_i$. For example, if α is an interaction task (i.e. an input action) labeled with *Any*, this means that the task is performed on exactly one of the defined devices.

Next, the semantics of the *All* operator is defined:

$$All^{c_1,\ldots,c_i}(\alpha)$$

denotes that the CTT α shall be executed on **all** of the devices c_1 to c_i. Likewise, if α is an interaction task labeled with *All*, the task's associated input action must be performed on all defined devices. An example could be to "connect a cable" on all devices before communication can start. Note that for one device label, *Any* and *All* are equivalent.

Now, consider the following operational semantics for *Any* on basic tasks:

$$Any^{c_1,\ldots,c_k}(a) \xrightarrow{c_i(a)} emCTT$$

where $1 \leq i \leq k$ and a is a basic task. This means that for task a, an arbitrary c_i can be chosen for its execution.

Similarly, the operational semantics for *All* on basic tasks is as follows:

$$All^{c_1,\ldots,c_k}(a) \xrightarrow{c_i(a)} All^{c_1,\ldots,c_{i-1},c_{i+1},\ldots,c_k}(a)$$
$$All^{c}(a) \xrightarrow{c(a)} emCTT$$

This expresses that the basic task a is executed on all devices. Note that the order in which the distinct devices execute a is irrelevant.

Next, the effects and semantics of *Any* and *All* applied to CTT operators are defined. The main idea is to push the *Any* and *All* operators inside, towards the leaves of the CTT, where basic tasks can be executed on the specified devices[2]. For an *Any*-annotated *Choice*, the following semantical rule applies:

$$Any^{c_1,\ldots,c_n}(Ch(\alpha_1,\alpha_2)) \longrightarrow Any^{c_i}(\alpha_k)$$

where $k \in \{1,2\}$. Consequently, the semantics of an *Any*-annotated *Enabling* operator is as follows:

$$Any^{c_1,\ldots,c_n}(En(\alpha_1,\alpha_2)) \longrightarrow En(Any^{c_i}(\alpha_1), Any^{c_i}(\alpha_2))$$

Note that both α_1 and α_2 are labeled with the same device label. This indicates that the device that was selected for the execution of α_1 must also be used for the execution of α_2.

The same semantical rule applies for the *Any*-annotated *Disabling* operator:

$$Any^{c_1,\ldots,c_n}(Di(\alpha_1,\alpha_2)) \longrightarrow Di(Any^{c_i}(\alpha_1), Any^{c_i}(\alpha_2))$$

Consequently, for the semantics of the *Concurrent* operator, the following rule applies:

$$Any^{c_1,\ldots,c_n}(Co(\alpha_1,\alpha_2)) \longrightarrow Co(Any^{c_i}(\alpha_1), Any^{c_i}(\alpha_2))$$

The semantical rule for the *All* operator is generic for all CTT operators. Basically, *All* is pushed inside the CTT α:

$$\begin{aligned}
All^{c_1,\ldots,c_n}(Ch(\alpha_1,\alpha_2)) &\longrightarrow Ch(All^{c_1,\ldots,c_n}(\alpha_1), \\
&\qquad All^{c_1,\ldots,c_n}(\alpha_2)) \\
All^{c_1,\ldots,c_n}(En(\alpha_1,\alpha_2)) &\longrightarrow En(All^{c_1,\ldots,c_n}(\alpha_1), \\
&\qquad All^{c_1,\ldots,c_n}(\alpha_2)) \\
All^{c_1,\ldots,c_n}(Di(\alpha_1,\alpha_2)) &\longrightarrow Di(All^{c_1,\ldots,c_n}(\alpha_1), \\
&\qquad All^{c_1,\ldots,c_n}(\alpha_2)) \\
All^{c_1,\ldots,c_n}(Co(\alpha_1,\alpha_2)) &\longrightarrow Co(All^{c_1,\ldots,c_n}(\alpha_1), \\
&\qquad All^{c_1,\ldots,c_n}(\alpha_2))
\end{aligned}$$

Finally, conflicting device labeling operators must be considered (e.g. *Any* immediately followed by *All* or vice versa). In this case, preference is always given to the most inner device labeling operator. Consequently, this leads to the semantical rule

$$Any^{c_1,\ldots,c_n}(All^{c'_1,\ldots,c'_m}(\alpha)) \longrightarrow All^{c'_1,\ldots,c'_m}(\alpha))$$

as well as

$$All^{c_1,\ldots,c_n}(Any^{c'_1,\ldots,c'_m}(\alpha)) \longrightarrow Any^{c'_1,\ldots,c'_m}(\alpha).$$

3.3 Examples

The following examples illustrate the above rules. The examples formalize among others the introductory examples and show how the actual runtime semantics could possibly look like. Note that there can be a large number of possible execution orders. However, we only show one (arbitrarily selected) order.

[2]We could also use a simpler rule to define *Any* by just selecting one device. However, we still need to push the *Any* operator to the leaves, as it is only defined for leaves.

The first MCTT configuration is intended to set the time on a device, but all actions are performed on exactly one of the devices:

$$Any^{c_1,\ldots,c_i}(En(SetHour, En(SetMinute, SetSeconds)))$$

This example now reduces as follows:

$$\begin{aligned}
&\ldots \xrightarrow{k(SetHour)} En(SetMinute, SetSeconds) \\
&\xrightarrow{k(SetMinute)} SetSeconds \\
&\xrightarrow{k(SetSeconds)} emCTT
\end{aligned}$$

where $k \in \{c_1,\ldots,c_i\}$.

Next, we show the pizza mode selection and confirmation on our microwave. We want to specify that a switch of devices is possible (but not necessary) during task execution:

$$En(Any^{c_1,\ldots c_i}(SelPizza), Any^{c_1,\ldots,c_i}(ConfSelection))$$

This reduces to

$$\begin{aligned}
&\ldots \xrightarrow{k(SelPizza)} ConfSelection \\
&\xrightarrow{j(ConfSelection)} emCTT
\end{aligned}$$

for some k and j (k and j might also be equal).

The next example is entering the time, which we want to be interruptible. However, the interruption should occur on the same device as the process was started:

$$\begin{aligned}
&Any^{c_1,\ldots,c_i}(Di(En(\\
&\quad SetHour, En(SetMinute, SetSeconds)), stopMicro))
\end{aligned}$$

this may e.g. execute to

$$\begin{aligned}
&\ldots \xrightarrow{j(SetHour)} Di(En(SetMinute, SetSeconds), stopMicro) \\
&\xrightarrow{j(stopMicro)} emCTT
\end{aligned}$$

Note that other executions are possible (even "stopMicro" not happening at all).

Finally, overruling the device selection, e.g. when preparation steps may occur on any device, but opening the microwave must happen on the microwave itself is formulated as

$$\begin{aligned}
&Any^{c_1,\ldots,c_i}(En(En(_,\ldots,En(_, \\
&\quad \ldots Any^{microwave}(OpenDoor))\ldots)
\end{aligned}$$

and may execute e.g. as follows:

$$\ldots \xrightarrow{k(\ldots)} \ldots \xrightarrow{microwave(OpenDoor)} \ldots \xrightarrow{k(\ldots)} emCTT$$

4. TOOL CHAIN IMPLEMENTATION

We have implemented the device labeling mechanism in a prototype tool chain that allows for the execution of a model-driven development approach using MCTT task models.[3] The framework covers all steps from designing the MCTT model to building working device UIs. To execute the MCTTs, we have developed a translation of MCTT models into device-specific state machines for each involved device. Our translation extends the algorithm in [31] from CTTs to MCTTs and is described in [30]. The actual algorithm is beyond the scope of this paper. The basic idea

[3] The tool chain is available from the authors and can be retrieved from https://github.com/MultiDeviceCTT/MCTT.

is that the statechart of each device whose label is present in the label set of an *Any* or *All* operator keeps track of all states and transitions which are actually executed on other involved devices. This implies that an event that might occur on any of the defined devices will result in state changes in all involved statecharts. As a result, the execution of an *Any* or *All* operator happens completely synchronized on all involved devices. This is mainly achieved by state transitions that are triggered by events on remote devices and a well-defined distributed execution model. In the following, we focus on this model and the tool chain concept which supports the development.

From a technical point of view, the framework is based on the well-known *Qt framework*[4] and especially on the *QML scripting language*[5] which is part of Qt. Our intention is to provide a rapid-prototyping framework that offers features typically available in embedded and mobile devices, e.g. interaction possibilities (soft- and (emulated) physical buttons), timers, GUIs and networking. We chose Qt over other frameworks because it allows us to easily build visually appealing and interactive user interfaces which are driven by state machines. Bundled with state machines from the developed algorithm, this allows for the design of cross-device scenarios which provide full user interaction and networking capabilities.

Figure 6 depicts the code generation and system architecture for multiple devices. A "device" executes a (device-specific) state machine, displays a GUI and communicates with other devices as indicated. The devices are connected to a server (which is denoted as *coordinator* in Figure 6) via WebSockets. All application-specific components (e.g. state machines or GUI glue code) are generated and can be extended by manually written QML code. The QML parts are executed by a client runtime, which provides networking and state machine APIs. The utilized technologies allow system designers a mostly declarative device description without the need of writing low-level code or compiling source files.

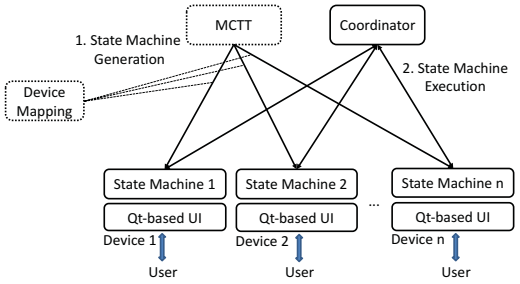


Figure 6: Overview of State Machine Generation and Distributed System Architecture

4.1 Implementation Overview

We basically split our prototyping environment into two main phases. In the first phase (i.e. *generation phase*), we generate the device-specific state machines from the MCTT. In essence, the developed algorithm creates a state machine for each device label present in the MCTT. The way how tasks are translated into state machine elements highly depends on the task type. In principle, interaction tasks are mapped to transitions whereas application tasks are mapped to states. Further, the algorithm generates additional transitions which are triggered when application tasks finish execution or remote devices are in charge of notifying the current local device. Therefore, the state machines also inherently cover communication between the devices. The resulting state machines are executed synchronously on the corresponding devices and implement the system behavior specified in the MCTT.

Because a device might only execute a subset of all the defined tasks (due to corresponding labelings in the MCTT), the algorithm generates state machines which are specifically tailored to the target device. However, an important feature of the generated state machines is that each device is aware of all states and transitions which are actually executed on other devices. The structure of the state machines of the distinct devices might therefore be similar, even if devices do not participate in the execution of a (sub) tree of the MCTT. As an example, consider that we label a subtree α of a MCTT with $Any^{x,y}$. If we now consider the generated state machine of a device z, which is not part of the labeling of α but of another subtree, the state machine will still have the states and transitions representing α. However, z will not perform tasks when its state machine is in one of those states. Instead, this part of z's state machine only consists of *dummy* states and transitions, which ensure that z is aware of the global system state. Due to proper selection of the state machine semantics, we thus achieve a synchronous execution of the state machines on all involved devices[6]. Consequently, all events that occur on any of the defined devices result in state changes on all state machines simultaneously.

4.2 UI Mapping

Another aspect of the generation phase is the mapping of tasks to actual device functionality. From an implementation point of view, a task is a rather abstract element which does not provide information on *how* it can be executed. Therefore, the framework needs to know which task is implemented by which device functionality (it may be a UI control or an internal function). The problem of linking abstract elements (in this case tasks) to concrete elements (their implementation) is called the "mapping problem" and is described in detail in [26]. For our evaluation purpose, we use a *mapping file*, inspired by the work of [7], which provides per-device information on the available UI controls and the task mapping itself (i.e. which task belongs to which UI control(s)). Further, it defines elements for more fine grained control over the mappings and action methods which provide actual device functionality and data to be transmitted to other devices.

A sample snippet of a mapping file is shown in Figure 7. It describes the configuration for a device named *deviceX*. *deviceX* has only three UI controls - *selectionCombo*, *confirmButton* and *powerButton*. The *visible* and *enabled* keywords are the control's *deactivation policies*. These are valid QML attributes and are set to true or false , depending on the current state.

The remainder of the file specifies task mappings. *taskA*, *taskB* and *taskC* are tasks the device should handle. For ex-

[4]https://www.qt.io/
[5]http://doc.qt.io/qt-5/qtqml-index.html

[6]In our implementation, we assume the STATEMATE execution semantics as described in [6].

```
[deviceX]
_controls = selectCombo:visible, confirmBtn:
    enabled
taskA = confirmBtn.clicked/selectCombo.currentText
    === "taskA"
taskB = confirmBtn.clicked
taskC = ^device.performTaskC(cb)
```

Figure 7: Sample mapping file entry for a device named "deviceX".

ample, *taskA* should be triggered by *confirmButton*'s click event, but only if *selectionCombo* displays the text "taskA". This *guard* is indicated by a slash ("/"). Finally, the task *taskC* is associated with the function performTaskC(cb) provided by the device. These *action methods* are indicated by a carat ("^"). The actual implementation of an action method is up to the system or device designer.

4.3 Distributed Coordination

The second phase in our prototyping framework is the *execution phase*. All communication between the devices is routed over the coordinator. If an event is generated on a device (e.g. because the user executes an interaction task), the device forwards the event to the coordinator, which in turn broadcasts the event to all other state machines in the system. Only after each device has received the event, the device which emitted the event is allowed to process it locally. If two devices emit events simultaneously, the coordinator decides which event should be broadcasted. Thus, race conditions can be avoided. Because of the way we generate the state machines, the event will lead to a state transition in *every* state machine. This concept is depicted in Figure 8.

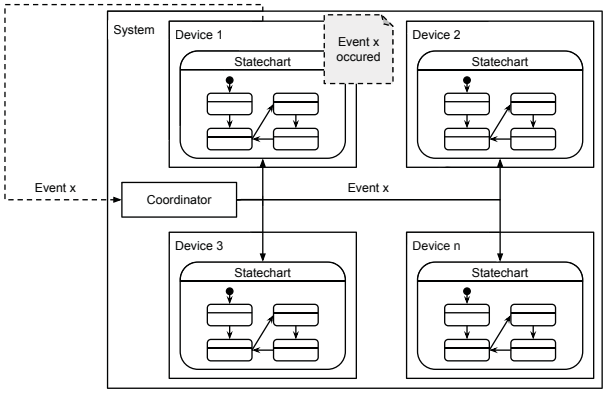


Figure 8: Global View on the System during the Execution Phase.

5. TV EXAMPLE WITH THREE DEVICES

Based on our implementation, we provide a sample scenario which employs the presented techniques. The scenario describes a TV, a remote control (abbreviated *RC*) and a smartphone. The idea is to use the smartphone as a second remote control. While the RC is a very limited device which only performs basic tasks, the smartphone should also provide an extended feature set.

The main goal of this scenario is to show how MCTTs can be used to implement concurrent tasks on several devices. Input actions can basically occur on any remote device and tasks (e.g. switching channels and changing volume) are not mutually exclusive. In the following, the development of the task model and the device UIs of TV, RC and smartphone are shown.

Table 2: Tasks of the TV example scenario.

Task name	**Description**
mute	mute TV
performMute	perform mute task
volumeUp	increase volume
performVolumeUp	perform volume increase
channelUp	next channel
performChannelUp	show next channel
⋮	
standBy	actually go to standby
softSwitchOn	turn TV on
returnFromStandBy	actually return from standby
hardSwitchOff	turn TV off with switch
hardSwitchOn	turn TV on with switch

5.0.1 Tasks and Task Model

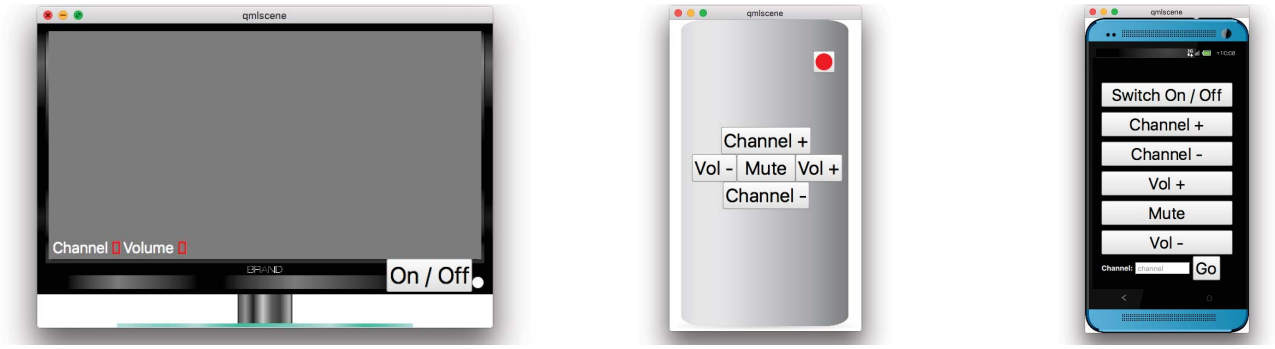
We use the typical basic tasks of a TV and remote control for our scenario. These include channel switching, adjusting the volume, muting and switching the TV on and off. The remote devices (RC and smartphone) are intended to perform the typical input actions like channel up and down or increasing and decreasing the volume. The TV is intended to be a passive device which performs the desired input actions. Smartphone and RC are devices with similar features, but the smartphone provides additional tasks like directly addressing a channel and a more involved UI.

For the task model to be successfully mapped on the different devices, tasks must be decomposed in an input and in a performing part. For example, switching to the next channel requires two actions. First, the task must be initiated by the RC or smartphone (task *channelUp*), then it must be performed by the TV (task *performChannelUp*). Altogether, this leads to the task set shown in Table 2.

In order to obtain sensible state machines, the tasks must be classified as *interaction* or *application* tasks. As a rule of thumb, tasks which are performed by remote devices (RC or smartphone) are interaction tasks. Tasks that are performed by the TV are application tasks. The only exception are the tasks *hardSwitchOff* and *hardSwitchOn*, which are interaction tasks performed by the TV.

Combining the tasks into a task tree can be done bottom-up. First, the decomposed tasks are put together by means of the *Enabling* operator, e.g. *(channelUp >> performChannelUp)*. These building blocks are then combined to bigger task trees. Note that the intended functionality requires the correct operators. For example, the user must be able to change the volume even if channels are selected on another device. On the other hand, switching channels must be possible even when the TV is muted. Therefore, we use the *Concurrent* operator to combine the task trees which define volume adjustment and muting with the task tree that defines channel selection.

For the TV to be turned off we introduce a new CTT operator called *Reset*. *Reset* is similar to the *Disabling* operator. It is a binary operator, whose left subtree α can be interrupted by the right subtree β at any time during the execution of α. The difference is that α is started all over again when β finished. Also, a new MCTT operator $Repeat(\alpha)$ has



(a) The TV device UI. (b) The remote control UI. (c) The smartphone UI.

Figure 9: Device user interfaces.

been introduced, which defines that the subtree α must be executed at least once. To be able to iteratively change channels and volume, the corresponding subtrees are wrapped in the *Repeat* operator.

Note that there are two ways to turn the TV off. The first way is to enable standby mode, which is modeled by means of the *Reset* operator. The second way is to turn it off physically, which - for a real TV - would mean to cut power supply. Turning the TV off physically can even interrupt the standby mode. We model this behavior by simply wrapping the whole task tree into another *Reset* operator. Its subtree β consists of an *Enabling* operator with tasks *hardSwitchOff* and *hardSwitchOn*.

For the MCTT model to be complete, tasks and/or subtrees must be labeled with devices. We model all interaction tasks (except *hardSwitchOff* and *hardSwitchOn*) as $Any^{rc,sp}$. All application tasks are labeled as Any^{tv}. Note that because the interaction tasks are intended to be executed by any remote device, we don't label whole task trees but only single tasks. This allows switching devices at any time (e.g. increasing volume on the smartphone while changing channels on the remote control).

The complete MCTT model is shown in Figure 12. Due to lack of space, we use a tree with the textual notation, not with the above graphical notation. What remains to be shown are the device user interfaces for TV, smartphone and RC and task implementation examples.

5.1 TV, Remote Control and Smartphone UIs

The TV can increase and decrease the volume, increment and decrement the current channel and set a channel directly. Channel and volume are displayed in "status displays", which are simulated LCD screens.

The system has two "off states", namely a standby mode and a physical off state. The TV can be physically turned off and on only by a corresponding (physical) button which disables the whole system. The TV user interface is depicted in Figure 9a.

As an example for the TV's task implementation, the task *standBy* is shown in Figure 10. It simply turns off the screen, enables the standby LED and calls a callback, indicating that task execution has finished. The callback returns control to the device's state machine. The next available steps are then determined by the state machine and the execution environment (both local and at remote devices). This is completely transparent to the system designer and keeps the implementation simple.

```
function standBy(cb){
    // turn off screen
    screen.color = "black"
    // enable standby LED
    standByLight.color = "red"
    // notify that task has finished
    cb()
}
```

Figure 10: Implementation of task *standBy*.

The design of the RC device only covers the basic features of the TV. These include adjusting the volume, switching channels, muting the TV as well as turning it on and off (standby). The feature list is inspired by remote controls actually available for the elderly [16][7], so this kind of reduced remote controls is indeed relevant in practice. The RC UI is depicted in Figure 9b.

Because the RC device does not implement application tasks, the actual device implementation reduces to just model the GUI. No additional methods are necessary to implement the device. All functionality comes purely from the MCTT, the task mapping (which can be seen in Figure 11) and the generated state machine.

The smartphone provides the same functionality as the remote control. However, the smartphone additionally allows setting the desired channel directly. Furthermore, the smartphone UI is instantly adapted to the available tasks or completely disabled if a hard turn-off on the TV occurred. Figure 9c shows the smartphone device UI.

```
[rc]
_controls     = onOffButton:enabled, ...,
                muteButton:enabled
volumeUp      = volPlusButton.clicked
volumeDown    = volMinusButton.clicked
channelUp     = chanPlusButton.clicked
channelDown   = chanMinusButton.clicked
softSwitchOn  = onOffButton.clicked
softSwitchOff = onOffButton.clicked
mute          = muteButton.clicked
unmute        = muteButton.clicked
```

Figure 11: Task mapping for the example scenario (remote control).

6. DISCUSSION

In this section, we discuss our approach and also outline some extensions for future work. Regarding CTTs one

[7]e.g. the "Tek Pal Universal Remote Control" http://www.bigbuttonremotes.com/remotes-tek-pal.htm

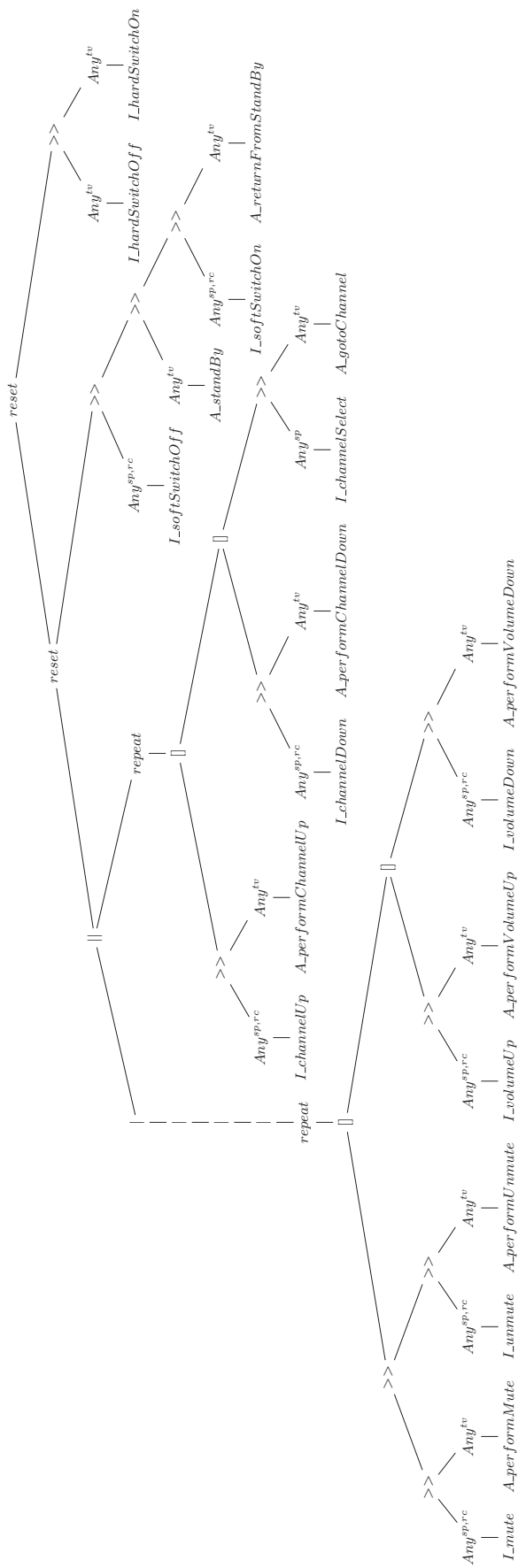


Figure 12: MCTT model for the example scenario.

question is the notation, and also the scalability in terms of model size. In the example we can recognize, that even small cases can lead to large diagrams (see Figure 12). Even though the example is using only binary CTTs, which increases the size, it becomes clear that some more concise, graphical notation as in Figures 1-4 with suitable tools support may be needed.

On the other hand, MCTTs model a complex, distributed system, resulting in automatically generated state machines and a generic broadcast communication mechanism based on events. This has proven to work nicely and takes a way considerable complexity from the developer. We should note here that the state machines for the UI must also be able to display, on all involved devices, which actions are currently possible on the current device. This is needed to present the possible actions as soft buttons and requires considerable analysis which tasks are executable next. As shown in [30, 31], this is complex due to the concurrent and disabling operators, where the currently possible interactions must be found in multiple leaves of the MCTTs. For scalability of the models as such, we essentially have the same issues as in basic CTTs and other modeling languages. For instance, the work in [10] discusses scalability and modularity, as well as relation and integration with other UML-based modeling languages and tools.

Another question is the expressiveness of the concepts. Our examples have shown that the operators *Any* and *All* can cover quite a wide range of cases. There are however two extensions which can be useful in practice. One is the addition of classes of devices, e.g. if several devices with identical behavior are present. An example would be two users with the same mobile app controlling the TV. Such an extension could simply extend the language of the devices to include classes and instances of devices. In the same line, we could consider cases where the number of devices changes over time. We expected that our formalism can easily be extended to this.

7. RELATED WORK

Task models are mainly used for model based user interface development (MBUID), and many researchers have investigated how task models support multi-device applications development in ambient intelligence environments. Paternò et al. connect web services with CTTs by associated annotations to develop service-oriented multi-device interactive applications. This approach exploits the web service annotation for model transformations at various abstract levels [24]. However, contrarily to our approach, designers have to design a distinct CTT for each device to connect them with the web services in order to develop different versions of the same application on multiple devices.

Besides, Melchior et al. use a distribution graph consisting of a state-transition diagram to distribute UIs to various devices. In the state-transition diagram, states represent significant distribution states of distributed user interfaces and transitions consist of event, condition and action [14]. However, creating a complete distribution graph for bigger scenarios is not easy and the graph quickly becomes complex. More generally, their approach does not consider distribution in the task model directly.

Luyten and Clerckx develop an algorithm for transforming a CTT to a set of ETSs (Enabled Task Sets) and define semantics for the temporal operators *Enabling* and *Disabling*

for the transition [11]. This is similar to the state machines used here, but does not consider multi-device environments.

Luyten et al. show a task-centered approach to design ambient intelligent applications. The task distributions are context-aware by introducing an environment model to describe the configuration of devices at particular time. This does however not cover the modeling of depencencies of tasks in a dedicated language nor the dynamic assignmet [12].

Wurdel et al. present the Collaborative Task Modeling Language (CTML) which is designed to model actors, roles, collaborative tasks and their dependency on the domain with precisely defined syntax and semantics. In order to support smart environments, CTML employs team modeling and device modeling modules and it enables designers to specify tasks for smart environments including device modeling and other contexts [32]. However, integrating CTML with device modeling does not concern tasks to be performed on multiple devices.

In order to reduce the complexity of multi-device application development, some researchers have leveraged model-driven engineering (MDE) methodology to develop multi-device applications [8, 9]. These do however not address the multi-device aspects directly.

Other literature on multi-device or distributed user interactions discusses the general concepts of complementary, redundant or equivalent user interface elements [4, 13]. Our goal is to formalize typical such patterns as extensions to CTTs. While other extensions for CTTs permit expressive constraints and dependencies on tasks [2, 28, 12], we devise here specific, well defined operators to express the spatial relations between tasks in a distributed scenario. For instance, with current techniques it is not easily possible to express that some task has to be executed on all devices, a single device or on a subset of devices of an overall set of devices.

Finally, [17, 18, 23] discuss many aspects of cross-device user interfaces, including migration of a user interface, timing aspects and adaptation to new devices.

Usually, all these existing approaches handle distribution issues in more concrete models after defining the task models. In our opinion, it is however natural to consider the distribution to devices in task models directly. After distributing tasks to particular devices, user interfaces or applications corresponding to the distributed tasks can be mapped to particular devices. In addition, instead of adding rules for executing tasks across multiple devices at the concrete model level, our introduced device labeling mechanism enables designers to define execution of tasks at the early stage. This decreases the overhead of defining additional distribution and execution models in later-on, concrete implementations.

8. CONCLUSION

In this paper, we have presented an extension to task models, called MCTTs, which adds the dynamic assignment of tasks to an arbitrary subset of devices and specifies the execution semantics of tasks on these device sets. While CTTs model the temporal and causal relations between tasks, MCTTs can also specify relations on the mapping of tasks to devices. This can be seen as an additional, independent dimension, compared to the temporal relations in usual CTTs. We integrate this into one new modeling language and have defined semantics for it.

The main novelty of the language is the introduction of the two new operators *Any* and *All*, to specify if a (composed) task should be executed on any or on all devices of a set of devices. We have shown that this is applicable in scenarios of connected, smart devices where similar or identical tasks can be executed on a multitude of devices. Because we treat these operators as regular CTT operators, we can place them arbitrarily within the task tree, which especially allows nested operators. This is useful in case of exceptions, e.g. when arbitrary devices can be selected for the execution of a complex task, but some inner task must always be mapped on a specific device.

Based on the semantics for our extension, we have implemented a tool chain with a mapping to distributed state machines based on the Qt toolkit. The tool chain includes basic mechanisms to map high-level tasks to concrete UI controls, as well as a distributed execution and coordination model. It has been validated in several case studies and one of them has been presented in more detail. In summary, we provide a new approach and tool chain to design multi-device applications by extending task models in an expressive and well-defined way.

Acknowledgments: This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645119.

9. REFERENCES

[1] Jens Brüning, Anke Dittmar, Peter Forbrig, and Daniel Reichart. 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 (2008), 175–192.

[2] Sybille Caffiau, Dominique Scapin, Patrick Girard, Mickaël Baron, and Francis Jambon. 2010. Increasing the expressive power of task analysis: Systematic comparison and empirical assessment of tool-supported task models. *Interacting with Computers* 22, 6 (2010), 569–593.

[3] Jacek Chmielewski. 2014. Device-Independent Architecture for ubiquitous applications. *Personal and Ubiquitous Computing* 18, 2 (2014), 481–488.

[4] Joelle Coutaz, Laurence Nigay, Daniel Salber, Ann Blandford, Jon May, and Richard M. Young. 1995. Four easy pieces for assessing the usability of multimodal interaction: the CARE properties. In *IFIP Conference on Human-Computer Interaction*. 115–120.

[5] Dagmawi L Gobena, Gonçalo NP Amador, Abel JP Gomes, and Dejene Ejigu. 2015. Delegation Theory in the Design of Cross-Platform User Interfaces. In *Human-Computer Interaction: Interaction Technologies*. Springer, 519–530.

[6] David Harel and Amnon Naamad. 1996. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology* 5, 4 (1996), 293–333.

[7] Gavin Kistner and Chris Nuernberger. 2014. Developing User Interfaces using SCXML Statecharts. *Workshop on Engineering Interactive Systems with SCXML, The sixth ACM SIGCHI Symposium on Computing Systems* (2014), 5–11.

[8] Philippe Le Parc, Amara Touil, and Jean Vareille. 2009. A model-driven approach for building ubiquitous applications. In *The Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies-UBICOMM 2009*. 324–328.

[9] Grzegorz Lehmann, Andreas Rieger, Marco Blumendorf, and Sahin Albayrak. 2010. A 3-layer architecture for smart environment models. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*. IEEE, 636–641.

[10] Víctor López-Jaquero and Francisco Montero. 2007. Comprehensive task and dialog modelling. In *International Conference on Human-Computer Interaction*. Springer, 1149–1158.

[11] Kris Luyten and Tim Clerckx. 2003. Derivation of a dialog model from a task model by activity chain extraction. *Interactive Systems. Design ...* (2003), 203–217. http://link.springer.com/chapter/10.1007/978-3-540-39929-2_14

[12] Kris Luyten, Jan Van den Bergh, Chris Vandervelpen, and Karin Coninx. 2006. Designing distributed user interfaces for ambient intelligent environments using models and simulations. *Computers & Graphics* 30, 5 (2006), 702–713.

[13] Marco Manca and Fabio Paternò. 2011. Extending MARIA to support distributed user interfaces. In *Distributed User Interfaces*. Springer, 33–40.

[14] Jérémie Melchior, Jean Vanderdonckt, and Peter Van Roy. 2011. A model-based approach for distributed user interfaces. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 11–20.

[15] José Meseguer and Grigore Roşu. 2007. The rewriting logic semantics project. *Theoretical Computer Science* 373, 3 (2007), 213–237.

[16] Jim T. Miller. 2013. Simple Television Remotes Designed for Seniors. (2013). http://www.huffingtonpost.com/jim-t-miller/television-remotes-designed-for-seniors_b_3371888.html

[17] Michael Nebeling, Theano Mintsi, Maria Husmann, and Moira Norrie. 2014. Interactive Development of Cross-device User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2793–2802. DOI: http://dx.doi.org/10.1145/2556288.2556980

[18] Michael Nebeling, Fabio Paternò, Frank Maurer, and Jeffrey Nichols. 2015. Systems and Tools for Cross-device User Interfaces. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15)*. ACM, New York, NY, USA, 300–301. DOI: http://dx.doi.org/10.1145/2774225.2777463

[19] Fabio Paternò. 2000. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag London. 196 pages.

[20] Fabio Paternò. 2002. ConcurTaskTrees: an engineered approach to model-based design of interactive systems. *The Handbook of Analysis for Human-Computer Interaction, Lawrence Erlbaum Associates* (2002), 483–500.

[21] Fabio Paternò. 2015. Design and Adaptation for Cross-Device, Context-dependent User Interfaces. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '15)*. ACM, New York, NY, USA, 2451–2452. DOI: http://dx.doi.org/10.1145/2702613.2706686

[22] Fabio Paternò and Carmen Santoro. 2003. A unified method for designing interactive systems adaptable to mobile and stationary platforms. *Interacting with Computers* 15, 3 (2003), 349–366.

[23] Fabio Paternò and Carmen Santoro. 2012. A logical framework for multi-device user interfaces. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 45–50.

[24] Fabio Paternò, Carmen Santoro, Lucio Davide Spano, and HIIS CNR-ISTI. 2010. User task-based development of multi-device service-oriented applications.. In *AVI*. 407.

[25] Jorge-Luis Pérez-Medina, Sophie Dupuy-Chessa, and others. 2007. A survey of model driven engineering tools for user interface design. In *Task Models and Diagrams for User Interface Design*. Springer, 84–97.

[26] A. Puerta and J. Eisenstein. 1999. Towards a general computational framework for model-based interface development systems. *Knowledge-Based Systems* 12, 8 (1999), 433–442.

[27] Roman Rädle, Hans-Christian Jetter, Mario Schreiner, Zhihao Lu, Harald Reiterer, and Yvonne Rogers. 2015. Spatially-aware or spatially-agnostic? Elicitation and Evaluation of User-Defined Cross-Device Interactions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.

[28] Jan Van den Bergh and Karin Coninx. 2004. Contextual ConcurTaskTrees: Integrating dynamic contexts in task based design. In *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*. IEEE, 13–17.

[29] Jan Van Den Bergh and Karin Coninx. 2007. From Task to Dialog Model in the UML. In *Task Models and Diagrams for User Interface Design*. 98 – 111. http://www.springerlink.com/content/lp53242720572662

[30] Andreas Wagner. 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.

[31] Andreas Wagner and Christian Prehofer. 2016. Translating Task Models to State Machines. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development*. 201–208. DOI: http://dx.doi.org/10.5220/0005681702010208

[32] Maik Wurdel, Christoph Burghardt, and Peter Forbrig. 2009. Making task modeling suitable for smart environments. In *Ultra Modern Telecommunications & Workshops, 2009. ICUMT'09. International Conference on*. IEEE, 1–6.