CEA Standard

Task Model Description CE Task 1.0

CEA-2018

November 2007

CEA
Consumer Electronics Association
www.CE.org

NOTICE

Consumer Electronics Association (CEA®) Standards, Bulletins and other technical publications are designed to serve the public interest through eliminating misunderstandings between manufacturers and purchasers, facilitating interchangeability and improvement of products, and assisting the purchaser in selecting and obtaining with minimum delay the proper product for his particular need. Existence of such Standards, Bulletins and other technical publications shall not in any respect preclude any member or nonmember of CEA from manufacturing or selling products not conforming to such Standards, Bulletins or other technical publications, nor shall the existence of such Standards, Bulletins and other technical publications preclude their voluntary use by those other than CEA members, whether the standard is to be used either domestically or internationally.

Standards, Bulletins and other technical publications are adopted by CEA in accordance with the American National Standards Institute (ANSI) patent policy. By such action, CEA does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the Standard, Bulletin or other technical publication.

This CEA Standard is considered to have International Standardization implication, but the International Electrotechnical Commission activity has not progressed to the point where a valid comparison between the CEA Standard and the IEC document can be made.

This Standard does not purport to address all safety problems associated with its use or all applicable regulatory requirements. It is the responsibility of the user of this Standard to establish appropriate safety and health practices and to determine the applicability of regulatory limitations before its use.

(Formulated under the cognizance of the CEA's **R7 Home Network Committee.**)

Published by

©CONSUMER ELECTRONICS ASSOCIATION 2007
Technology & Standards Department
1919 S. Eads Street
Arlington, Virginia 22202

PRICE: Please call Information Handling Services, USA and Canada (1-800-854-7179)
International (303-397-7956), or
http://global.ihs.com

FOREWORD

This standard was developed under the auspices of the Consumer Electronics Association (CEA) R7 Home Network Committee.

All of the RelaxNG schemas and XML examples in this document have been automatically inserted from files and mechanically checked for syntactic validity.

First time readers are advised to first read the informative introduction in 6.

Contents

Page

Figures

Tables

CEA-2018
Task Model Description
CE TASK 1.0

1 SCOPE

A task model is a formal description of the activities involved in completing a task, including both activities carried out by humans and those performed by machines. This standard defines the semantics and an XML notation for task models relevant to consumer electronics devices. The standard does not depend on any specific home networking technology or infrastructure.

2 CONFORMANCE (Informative)

Figure 1 illustrates the conformance approach of this standard. At the center of the diagram is a *task model description*, which is an XML document. A task model description is used by a *task-based application* to guide its interaction with a user. A task model description specifies task classes, and representations of their intent and how high-level tasks can be decomposed into lower-level tasks. A task model description also contains ECMAScript programs which ground primitive tasks to devices via particular networking platforms. This standard defines an XML language for task model descriptions which is independent of task-based applications. See 6 for a further informative introduction tasks and task modeling.

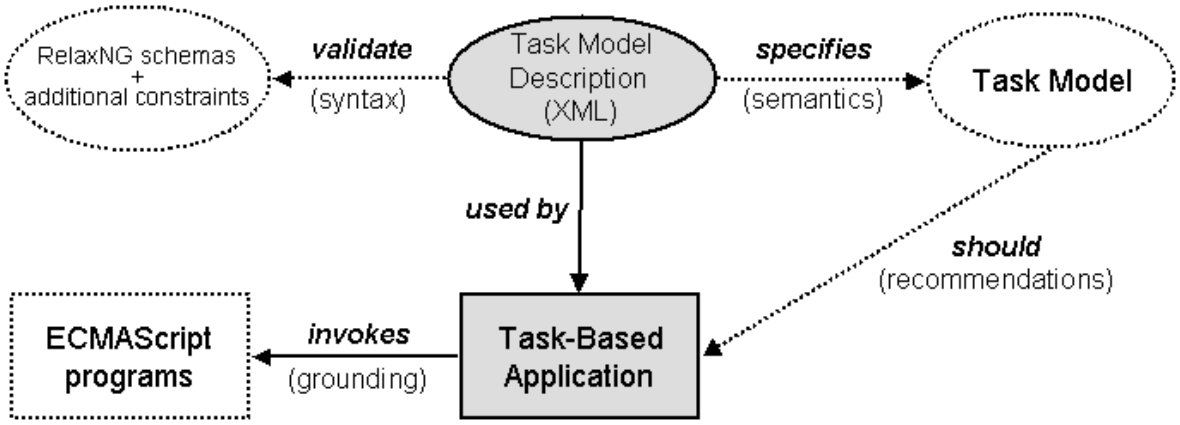

Figure 1: Conformance to This Standard

In general, the process of writing (authoring), distributing and using a task model description involves various stakeholders.

There are many different options for writing and distributing a task model description. First, a device manufacturer may write a task model description for their device, and either build it into the device so that it can be retrieved by a task-based application (see [REF 10.2.4]) or distribute it through their web server. Second, the developer of a task-based application may build task model descriptions into their application, or more likely, provide them through their web server. Finally, a third party may write task model descriptions for any device and make it available publicly through a web server.

Task model descriptions from different sources may mutually reference each other. For example, some task model descriptions may serve as “libraries” for specific domains. Similarly, the author of the ECMAScript program used to ground a particular task class (for a particular platform) may be different from the author of the task class specification.

2.1 Task Model Description

Conformance of task model descriptions is relevant to authors of task model descriptions, because it allows a task model description to be used by any task-based application based on this standard.

An XML document is a valid task model description if it conforms to the requirements outlined in 7 through 10 of this standard.

The syntactic validity of task model descriptions is completely and unambiguously established by the RelaxNG schemas in this standard, including the comments added to express restrictions that are not expressible in the schema language. (See Annex A for complete schemas.)

The semantics of task model descriptions is established by describing in words how a given document *specifies* a task model. A *task model* is an abstract *mathematical* construct, which is defined in this standard using common notions such as class, instance, set, function, etc.

2.2 Task-Based Application

Conformance of task-based applications is relevant to implementers of task-based applications, because that want their applications to make proper use of conformant task model descriptions.

The behavioral connection between a task-based application and the task model description it uses is established indirectly via the *recommendations* in 11. These are only recommendations, rather than mandatory requirements, because of the great variation possible in task-based applications. For example, this standard does not prevent an application from ignoring some aspect of the task model under direct instructions from the user.

2.3 Grounding

Conformance of the ECMAScript programs used for grounding is relevant for implementers of network platforms and devices, because it allows their platforms and devices to be used by task-based applications. The ECMAScript programs conform to this standard if they obey the specifications in 10.

3 REFERENCES

3.1 Other Standards Used

3.1.1 RelaxNG

The RelaxNG [6] schema language is used for specifying the XML syntax of task model descriptions.

NOTE See [14] for a short introductory tutorial on RelaxNG.

3.1.2 ECMAScript

ECMAScript [5] (commonly known as JavaScript) is used for grounding task model descriptions.

3.2 Normative References

The following standards contain provisions that, through reference in this text, constitute normative provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed.

[1] Extensible Markup Language (XML) 1.0 (Fourth Edition) – W3C Recommendation 16 August 2006, edited in place 29 September 2006, http://www.w3.org/TR/2006/REC-xml-20060816/

[2] IETF RFC 2046, Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types, Nov. 1996, http://www.ietf.org/rfc/rfc2046.txt

[3] IETF RFC 3986, Uniform Resource Identifier (URI): Generic Syntax, January 2005, http://www.ietf.org/rfc/rfc3986.txt

[4] ISO/IEC 10646:2003, Universal Multiple-Octet Coded Character Set (UCS)

[5] ISO/IEC 16262:2002, Information technology - ECMAScript language specification (also available from ECMA as Standard ECMA-262, ECMAScript Language Specification, 3rd edition, Dec. 1999)

[6] ISO/IEC 19757-2:2003, Information technology – Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG.

[7] Namespaces in XML – W3C Recommendation 14 January 1999, http://www.w3.org/TR/1999/REC-xml-names-19990114/

3.3 Informative References

[8] Dublin Core Metadata Element Set, Version 1.1, 2006-12-18, http://www.dublincore.org/documents/2006/12/18/dces/

[9] DCMI Metadata Terms, 2003-11-19, http://dublincore.org/documents/2003/11/19/dcmi-terms/

[10] Harley, Heidi. Thematic Roles. In: Patrick Hogan (ed.), The Cambridge Encyclopedia of Linguistics. Cambridge University Press, 2007.

[11] ISO/IEC FDIS 24752-1: Information Technology – User Interfaces – Universal Remote Console — Part 1: Framework. To be published in 2008

[12] ISO/IEC FDIS 24752-4: Information Technology – User Interfaces – Universal Remote Console — Part 4: Target Description. To be published in 2008

[13] Merriam-Webster Online, 2007. http://www.m-w.com/dictionary/task

[14] OWL Web Ontology Language Overview. W3C Recommendation 10 Feb 2004. http://www.w3.org/TR/2004/REC-owl-features-20040210/

[15] RELAX NG Compact Syntax Tutorial, Working Draft 26 March 2003. OASIS, 2003. http://www.relaxng.org/compact-tutorial-20030326.html

[16] “Thematic roles”. In: Santorini, Beatrice, and Anthony Kroch, The syntax of natural language: An online introduction using the Trees program, 2007. http://www.ling.upenn.edu/~beatrice/syntax-textbook/box-thematic.html

[17] UPnP™ Device Architecture 1.0, version 1.0.1, 2 December 2003, UPnP forum, http://www.upnp.org/resources/documents/CleanUPnPDA101-20031202s.pdf

[18] “Working Through Task-Centered System Design”, S. Greenberg, in *The Handbook of Task Analysis for Human-Computer Interaction*, D. Diaper and N. Stanton (eds), Lawrence Earlbaum, 2004, pp. 49-65. (See http://grouplab.cpsc.ucalgary.ca/papers/2002/02-TaskAnalysis.Chapter/task-analysis.chapter.pdf)

3.4 Reference Acquisition

ANSI/CEA Standards:

Global Engineering Documents, World Headquarters, 15 Inverness Way East, Englewood, CO USA 80112-5776; Phone 800-854-7179; Fax 303-397-2740; Internet http://global.ihs.com; E-mail global@ihs.com

DCMI Documents:

Dublin Core Metadata Initiative (DCMI); Internet http://www.dublincore.org

ECMA Documents:

ECMA International, Rue du Rhone 114, CH-1204 Geneva; Phone +41 22 849 6000; Fax +41 22 489 6001; Internet http://www.ecma-international.org/. ECMA Standard Documents can be downloaded electronically and free of charge at http://www.ecma-international.org/publications/standards/Standard.htm

IETF Documents:

Internet Engineering Task Force (IETF) Secretariat, c/o Corporation for National Research Initiatives, 1895 Preston White Drive, Suite 100, Reston, VA 20191-5434, USA; Phone 703- 620-8990; Fax 703-620-9071; Internet www.ietf.org, IETF RFCs may be downloaded from www.ietf.org/rfc.html, IETF Internet drafts may be downloaded from www.ietf.org/ID.html

ISO/IEC Documents:

International Organization for Standardization (ISO); ISO Central Secretariat; 1, rue de Varembé; Case postale 56; CH-1211 Geneva 20; Switzerland; Telephone +41 22 749 01 11; Fax +41 22 733 34 30; Internet http://www.iso.org/; ISO/IEC Standard Documents can be purchased and downloaded electronically at http://www.iso.org/iso/en/prods-services/ISOstore/store.html

UPnP Documents:

UPnP Forum; Internet http://www.upnp.org/

World Wide Web Consortium Documents:

World Wide Web Consortium (W3C); Internet http://www.w3.org/

4 TERM USAGE

Notwithstanding the placement of a NOTE or EXAMPLE in a normative section, paragraphs marked as “NOTE” or “EXAMPLE” shall be treated as informative.

4.1 Applications

DEFINITION 1 A ***task-based application*** (**“**application” for short*)* is a program that interprets task model descriptions according to the semantics defined in this standard.

For brevity, in the remainder of this document, a task-based application will be referred to generically as an *application*.

4.2 Use of Shall, Should and May

Specific keywords are used in this document to differentiate levels of requirements and optionality, as follows:

Shall: Indicates a mandatory requirement. Designers are required to implement all such mandatory requirements to assure interoperability with other products conforming to this standard.

Should: Indicates flexibility of choice with a strongly preferred alternative. Equivalent to the phrase “is recommended.”

May: Indicates flexibility of choice with no implied preference.

4.3 Schemas

The valid syntax for task model descriptions is defined using RelaxNG [6], with comments added to express restrictions that are not expressible in this schema language. These schemas are presented in Table 1 through Table 7 and are normative in this standard.

The syntax for the ’about’ document (see 10.2.3) is also defined using RelaxNG. The corresponding Table 9 is normative in this standard.

Notice that the order of XML sibling elements is fixed throughout the syntax schemas (attributes may be provided in any order). The motivation for this approach is to make it easier for humans to read task model descriptions and for applications to process them.

4.4 Namespaces

This standard defines and uses the XML namespaces "http://ce.org/cea-2018" for task model descriptions. Examples in this standard assume that this namespace is declared as the default namespace on the root element of a task model description document, such as in the following:

```
<taskModel xmlns="http://ce.org/cea-2018" …>
```

Task model descriptions may adopt this approach, or define an explicit namespace prefix. For other namespaces, the following three prefixes are used in the examples throughout this standard. Task model descriptions are not required to use these same prefixes as long as they declare the relevant namespaces properly (as specified in [7]).

- *xsd* -- the namespace http://www.w3.org/2001/XMLSchema (XML Schema Definition)
- *dc* -- the namespace http://purl.org/dc/elements/1.1 (DCMI Element Set). A schema definition file for this namespace is available at http://www.dublincore.org/schemas/xmls/qdc/2006/01/06/dc.xsd
- *dcterms* -- the namespace http://purl.org/dc/terms (DCMI Metadata Terms). A schema definition file for this namespace is available at http://www.dublincore.org/schemas/xmls/qdc/2006/01/06/dcterms.xsd

Namespace URIs, such as those above, are primarily for identification purposes and are not required to be resolvable [7], i.e., they might not serve a document when used as an internet address.

4.5 Classes and Instances

The standard terminology of object-oriented programming and design is used in this document without formal definition. In particular, a *class* typically has many *instances*, which differ in the values of the instance variables, or *slots* (as they are called in this document).

DEFINITION 2 A ***slot*** is a named, typed instance variable.

A class determines the *names* of the slots and the *type* of values each slot may hold; the slot *values* are determined by each instance. Finally, the class itself may have *properties* (sometimes called class variables), which are “inherited” by all instances, i.e., all instances have the same value for these properties.

4.6 Functions

DEFINITION 3 A ***partial function*** is a function which may be undefined for some argument values.

DEFINITION 4 An undefined ***function*** is a partial function which is undefined for all argument values.

5 DEFINITIONS

For the purposes of this document, the following terms and definitions apply. (This section lists the definitions of this standard in one place.)

DEFINITION 1 A ***task-based application*** (**“**application” for short*)* is a program that interprets task model descriptions according to the semantics defined in this standard.

DEFINITION 2 A ***slot*** is a named, typed instance variable.

DEFINITION 3 A ***partial function*** is a function which may be undefined for some argument values.

DEFINITION 4 An ***undefined function*** is a partial function which is undefined for all argument values.

DEFINITION 5 A ***task model*** is an ordered set of task classes, decomposition classes and scripts. The empty set is a valid task model.

DEFINITION 6 A ***task class*** is a class whose properties include a precondition, a postcondition, a (possibly empty) ordered set of user intent concepts and a (possibly empty) set of scripts.

DEFINITION 7 A ***task occurrence*** is a task instance which corresponds to an actual event.

DEFINITION 8 A ***hypothetical task instance*** is a task instance which does not correspond to an actual event.

DEFINITION 9 An ***input slot*** is a slot of a task class which is expected to have a value *before* execution of a task instance.

DEFINITION 10 An ***output slot*** is a slot of a task class which is not expected to have a value until *after* execution of a task instance.

DEFINITION 11 A ***user intent concept*** is a case frame, consisting of a verb and a set of semantic roles of specified types.

DEFINITION 12 A ***precondition*** is a partial boolean function property of a task class with arguments corresponding to the input slots of the task class.

DEFINITION 13 A ***postcondition*** is a partial boolean function property of a task class with arguments corresponding to the input and output slots of the task class.

DEFINITION 14 A ***decomposition class*** is a class whose properties include a step order, applicability condition and binding set.

DEFINITION 15 An ***applicability condition*** is a partial boolean function property of a decomposition class or a script with arguments corresponding to the input slots of the goal or script task class.

DEFINITION 16 A ***binding set*** is a (possibly empty) set property of a decomposition class containing bindings.

DEFINITION 17 A ***binding*** is an equality between an input slot of a decomposition step or an output slot of a decomposition goal, and the value of a function with arguments corresponding either to the output slots of steps or the input slot of the goal.

DEFINITION 18 A ***script*** is an ECMAScript [5] program which may be associated with one or more tasks classes, platforms and device types and whose properties include an applicability condition.

DEFINITION 19 An ***initialization script*** is a script which is not associated with any task class and is intended to be executed exactly once.

6 INTRODUCTION (Informative)

6.1 Tasks

The concept of task is at the heart of this standard. A dictionary [10] definition of task is:

> “a usually assigned piece of work often to be finished within a certain time.”

The task concept is also suggested by synonyms like "activity", "goal", "job", or "action". Examples of tasks in the domain of consumer electronics include copying a videotape to a DVD, watching a recorded TV episode, and turning off room lights. Tasks vary widely in their time extent: some take place over minutes or hours, e.g., watching a recorded TV episode; some are effectively instantaneous, e.g., turning off the room lights; and some have unbounded time extent, e.g., a weekly teleconference.

Tasks typically involve both human participants (e.g., as requesters, beneficiaries, or performers) and electronic devices. Some tasks may be performable only by a human being (e.g., providing fingerprint identification); others may be performed only by an electronic device (e.g., displaying a video); yet others may be performed by either, depending on the circumstances (e.g., opening the DVD drawer).

Tasks also vary along an abstraction spectrum from what might be called "high-level", i.e., closer to the user's intent and natural way of communicating, to "low-level", i.e., closer to the primitive controls of a particular device. Watching a recorded TV episode is an example of a fairly high-level task. Pressing the power button on a DVD player is an example of a very low-level task. Tasks are also more or less abstract by virtue of being parameterized. For example, watching the Season Two opening episode of The West Wing is a more specific version of the generic task of watching a recorded TV episode (in which the episode to be watched has been bound).

Specification of regularly scheduled tasks is out of the scope of this standard, since it can be very complicated.

6.2 Task Decomposition

High-level tasks usually need to be (repeatedly) decomposed into increasingly lower-level tasks (called *subtasks*) in order to accomplish them. This decomposition can sometimes be achieved entirely by the system; sometimes a collaboration between the system and user is required. The two main reasons why user involvement in task decomposition is sometimes required are:

- There are preferences involved which the user may need/want to provide. For example, when making a digital recording, a choice needs to be made in the fundamental tradeoff between picture quality and recording size. The system can help the user by explaining the options and their implications, and suggesting defaults. Similarly, the user may help in identifying which device should be used for the execution of a task, if multiple devices are available that can be used.
- The hardware may not be physically capable of performing certain actions, such as putting a blank DVD into the recorder.

6.3 Task Classes and Instances

A task model, as we will see below, contains task *classes*. For example, pressing the power button on a DVD player is an example of a task class. Slots of this class might include who pressed the button, which DVD player was involved, and when the action took place. Thus David Smith pressing the power button on the DVD player in his living room at 3:15pm on January 1, 2006 is an example of an *instance* of this class. A task instance may correspond to an actual event or it may be hypothetical. See 8.3 for the ECMAScript representation of task instances used in this standard.

Applications typically manipulate both task classes and instances. This standard does *not*, however, specify a representation for task instances – that is up to application implementations.

Figure 2 and the definitions in 8 summarize this terminology for tasks. For brevity, this standard sometimes simply uses the term *task* below where the context makes it clear whether a task class or a task instance is meant, or the remark applies to both classes and instances.

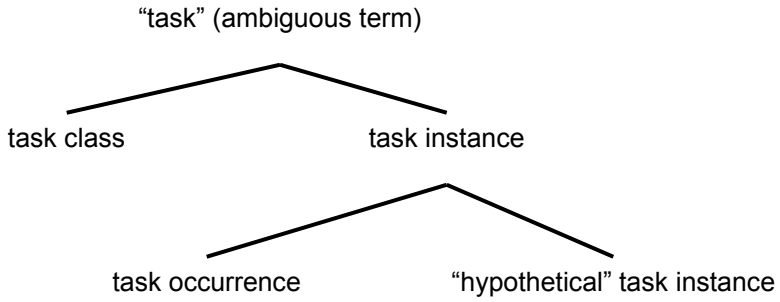

Figure 2: Task Terminology

6.4 Task-Based Applications

The motivation for defining this standard is to facilitate the development of task-based applications. Figure 3 shows the system architecture that this standard assumes. In a task-based application, the user and system interact primarily in terms of high-level goals, which the system decomposes into primitive actions that are directly supported by the one or more devices involved. Task-based applications address the problem that many people find new digitally-enhanced consumer electronic products too complex to operate and therefore do not use most of the features of them.

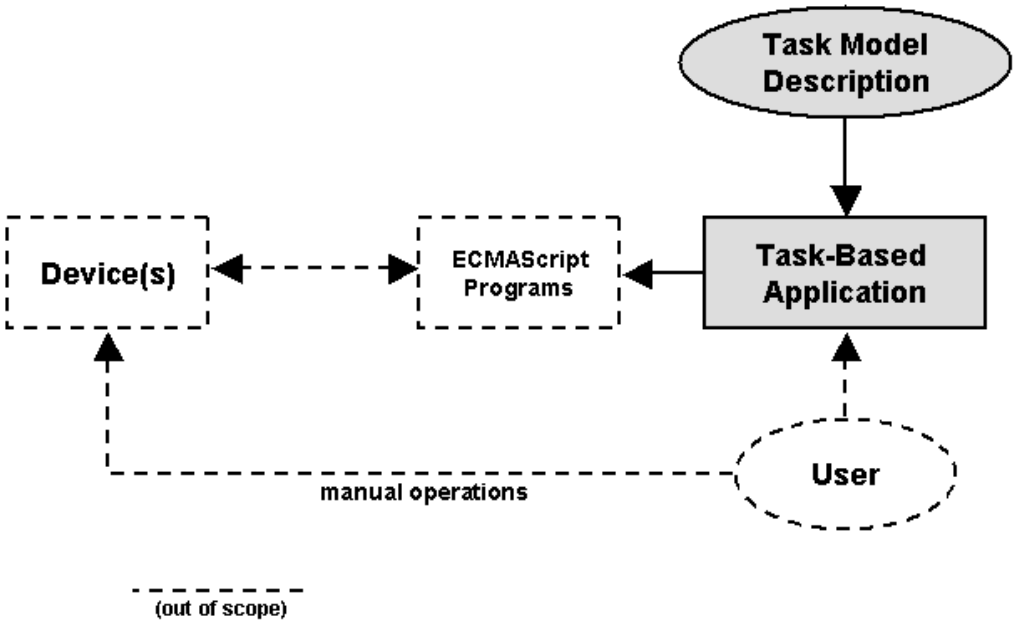

Figure 3: Expected Architecture for Using CEA-2018

Figure 3 is a functional, as opposed to physical, architecture. For example, the task-based application component may in fact run as a computational process on the same hardware as the device, or they may each have separate hardware. Furthermore, the display function of the user interface (if any) may share hardware with the device (e.g., if the device is a TV), or it may be a remote display.

Two fundamental inputs to the task-based application are the *task model description* (which is formalized in standard) and interaction with the user. Notice that the user interaction is out of the scope of this standard.

Communication between the task-based application and the device(s) is expected to be achieved via some kind of underlying network platform, such as UPnP. The framework for this communication is defined in 10 on grounding. The specific contents of this communication depend on the task models used.

Notice that this standard also allows for the possibility that the user will perform manual operations on the device, such as loading a CD. In fact, some devices may only be operable manually (e.g., they are not connected to the network), in which case the user interface would simply be providing instructions to the user for what to do. Manual operations are also out of the scope of this standard.

6.5 Task Engines

Figure 4 shows a possible refinement of Figure 3, in which the task-based application has been implemented in two parts: a generic *task engine* and an application-specific user interface which uses the task engine. Notice that all of this refinement is outside the scope of this standard.

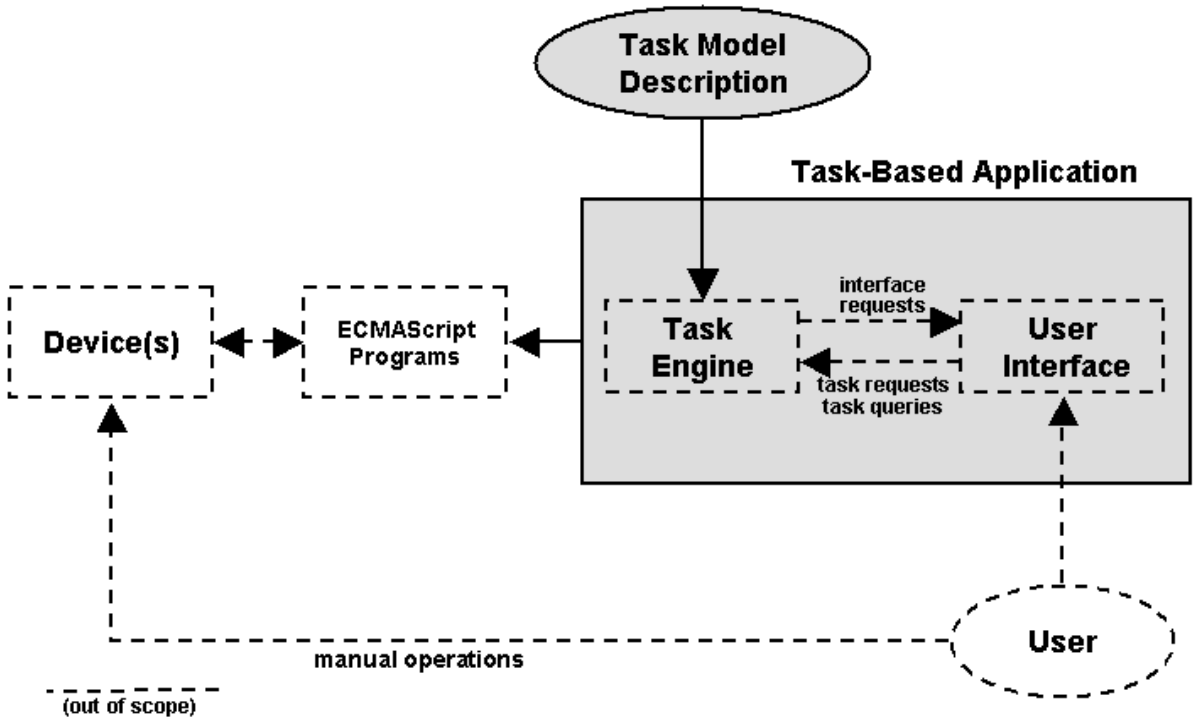

Figure 4: Possible Refinement of Figure 3 with Generic Task Engine

The basic functions of the task engine are to load the task model description and to maintain a representation of the current status of the user's task instances (see 6.3). The advantage of this approach is that the effort of building a task engine can be amortized over many different task-based applications. Notice that the task engine does not interact directly with the user. If the task engine needs input from the user, it sends an *interface request* to the user interface; how this request is presented to the user depends on the specific user interface, e.g., it may be visual, spoken, etc. Furthermore, the task engine provides an API through which the user interface may access the current task state (*task queries*) or cause a certain task to be performed (*task requests*).

Finally, like Figure 3, this is a functional architecture---the task engine and user interface may or may not be part of the same physical component, and the implementation of the communication between them depends on the situation.

6.6 Task Generation and Recognition

Two key functions typically performed by task-based applications are task generation and task recognition.

Task generation, sometimes called planning, is the process by which the system decides what it should do next. There are two different techniques typically used for this purpose, which may be intermixed. This standard is intended to support both techniques.

One task generation technique, called *first-principles planning*, searches through the set of all possible sequences of primitive actions to find a sequence which transitions from the current world state to the goal state. First-principles planning is particularly important for recovering from anticipated error conditions. First-principles planning requires a complete formalization of the preconditions and postconditions of each primitive action (see 8.4 and 8.5).

The other common task generation technique, called *hierarchical task networks*, uses a set of task decomposition rules to recursively decompose high level goals into primitive actions. This technique does not require as rigorous task modeling as first-principles planning.

Task recognition is the process by which the system observes the actions of the user and infers from them the user's possible intent (sometimes called goal recognition) and what method the user has chosen to achieve it (sometimes called plan recognition). Task recognition often reduces the communication burden on the user.

6.7 Task Modeling

A full discussion of task modeling is beyond the scope of this document (see [18] for a brief introduction and pointers into existing literature). However, a few general comments may be helpful here.

Most of the tasks that are defined with this standard do *not* have pure mathematical definitions. Instead, this standard is intended to be used to describe activities in the "real world" of people's living rooms, cars, etc. A big issue in writing such task descriptions is how accurately they correspond to "reality". For example, have all the inputs and outputs been accounted for (see 8.1)? Are the preconditions and postconditions complete (see 8.4 and 8.5)?

Unfortunately, such questions do not have a correct answer in general. In principle, the more accurately tasks are modeled, the better applications can perform using these models. On the other hand, highly accurate modeling can be very laborious and open-ended. It usually doesn't pay to put a lot of effort into a more accurate model than is required for a particular application.

EXAMPLE As a simple example, consider the following specification of a low-level task that is intended to model "turning the power on." Note that the <dc:description> element is used to annotate the task with id “powerOn” internally. In general, the intended audience of <dc:description> elements is the task model authors, not end users.

```
<taskModel
        about="http://ce.org/cea-2018/descriptionExample"
        xmlns="http://ce.org/cea-2018"
        xmlns:dc="http://purl.org/dc/elements/1.1">
    <task id="powerOn">
        <dc:description>Turning the power on.</dc:description>
    </task>
</taskModel>
```

For some purposes, this simple specification – really just an arbitrary identifier 'powerOn' with a human-readable description (using the namespace of the Dublin Core Metadata Element Set [8]) – is all that will be needed. In other contexts, more elaborate modeling, such as including preconditions and postconditions, may be needed.

A task modeling language, such as the one defined by this standard, shares some features with programming languages. At a fundamental level, both are finite representations which are executed/interpreted to produce potentially unbounded computations. Both are also intended to be authored by humans (more or less easily depending on the quality of the tools used).

However, applications do not only execute task models. Applications also need to explain them, recognize them, learn them, modify them, and generally reason about them. The design of this standard is therefore influenced by artificial intelligence knowledge representation and planning formalisms, as well as programming languages.

7 TASK MODEL DESCRIPTION

7.1 MIME Type

A task model description shall have a MIME type of “application/cea-2018+xml”, (with no parameters) if applicable [2].

7.2 Format and Encoding

A task model description shall be an XML 1.0 document [1].

The XML document shall use the Universal Multiple-Octet Coded Character Set (UCS) [4] (also known as “Unicode”).

The XML document shall be encoded in UTF-8.

7.3 XML Namespace

The XML elements and attributes defined in this standard for the specification of a task model description shall be of the XML namespace [7] "http://ce.org/cea-2018", except for the ‘about’ document (see 10.2.3) which shall be of the namespace "http://ce.org/cea-2018/about".

NOTE This namespace URI is also specified in the RelaxNG schemas in this standard (Tables 1 through 7, and Annex A).

7.4 Root Element

DEFINITION 5 A ***task model*** is an ordered set of task classes, decomposition classes and scripts. The empty set is a valid task model.

```
default namespace = "http://ce.org/cea-2018"

namespace xsd = "http://www.w3.org/2001/XMLSchema"
namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
namespace dc = "http://purl.org/dc/elements/1.1"
namespace dcterms = "http://purl.org/dc/terms"

start =
    element taskModel {
        # about shall be non-empty and not contain fragment identifier
        attribute about { xsd:anyURI },
        Description*,
        ( Task | Subtasks | Script )*
    }

Description =
    element dc:* { attribute * { text }*, text? } |
    element dcterms:* { attribute * { text }*, text? }
```

Table 1: RelaxNG Schema for Task Model

The root element of a valid task model description shall be <taskModel>, which specifies a task model containing the task classes, decomposition classes and scripts specified by its nested elements (see 8, 9 and 10) in the order in which the nested elements appear. See 11.2 for application recommendations regarding the ordering of task model members.

The 'about' attribute of a <taskModel> element shall be a universal resource identifier (URI) [3], which identifies the namespace in which the values of the ‘id’ attributes of the nested elements shall be defined. This URI shall not be empty, shall not contain a fragment identifier and may be not resolvable. A task model description (as well as <task> elements, see next section) may contain toplevel descriptive elements from the Dublin Core [8][9].

Each nested <task> element (see 8) in a <taskModel> shall have a unique 'id' attribute, which is the name by which the specified task model member shall be referred to in the namespace of the ‘about’ attribute.

EXAMPLE The following is a valid, but trivial, task model description that specifies a task model containing three task classes. See 9 for an example of how to refer to the members of this model in another task model description.

```
<taskModel
        about="http://example.com/myproject"
        xmlns="http://ce.org/cea-2018">
    <task id="X"/>
    <task id="Y"/>
    <task id="Z"/>
</taskModel>
```

8 TASK

DEFINITION 6 A ***task class*** is a class whose properties include a precondition, a postcondition, a (possibly empty) ordered set of user intent concepts and a (possibly empty) set of scripts.

DEFINITION 7 A ***task occurrence*** is a task instance which corresponds to an actual event.

DEFINITION 8 A ***hypothetical task instance*** is a task instance which does not correspond to an actual event.

```
Task =
    element task {
        attribute id { xsd:ID },
        Description*,
        Concept*,
        Input*,
        Output*,
        element precondition {
            # text shall be ECMAScript expression returning boolean or undefined
            # and may reference $this global variable
            text
        }?,
        element postcondition {
            [ a:defaultValue="false" ]
            attribute sufficient { xsd:boolean }?,
            # text shall be ECMAScript expression returning boolean or undefined
            # and may reference $this global variable
            text
        }?,
        element subtasks { SubtasksContent }*,
        element script { ScriptContent }*
    }
```

Table 2: RelaxNG Schema for Task

A <task> element specifies a task class uniquely identified by its ‘id’ attribute, with predefined slots (see 8.1.2) and other slots as specified by the nested <input> and <output> elements, and precondition, postcondition, user intent concepts and scripts properties as specified by the nested <precondition>, <postcondition>, <concept> and <script> (see 10) elements, respectively. The only required element or attribute in a <task> element is ‘id’. A <task> may contain toplevel descriptive elements from the Dublin Core [8][9].

EXAMPLE The following is a valid, but trivial, specification of a task class.

<task id="X"/>

A <subtasks> element nested in a <task> element specifies a decomposition class with the task specified by the enclosing <task> element as its goal and other properties as described in 9.

A <script> element nested in a <task> element specifies a script associated with the task specified by the enclosing <task> element and other properties as described in 10.

8.1 Input and Output Slots

DEFINITION 9 An ***input slot*** is a slot of a task class which is expected to have a value *before* execution of a task instance.

DEFINITION 10 An ***output slot*** is a slot of a task class which is not expected to have a value until *after* execution of a task instance.

```
Input =
    element input {
        Slot,
        # modified shall be the name of an output slot
        attribute modified { SlotName }?
    }

Output =
    element output { Slot }

Slot =
    # two slots within a given task shall not have the same name
    attribute name { SlotName },
    # type shall be primitive ECMAScript datatype or constructor defined
    # in current ECMAScript execution environment
    attribute type { DataType }

SlotName =
    xsd:NCName { pattern = "[^.\-]*" } - ("external" | "when" | "success")

DataType =
    "number" | "string" | "boolean" | xsd:token { pattern = "(\i|$)([\c-[:.\-]]|$)*" }
```

Table 3: RelaxNG Schema for Input and Output

Each <input> or <output> element specifies an input or output slot (respectively) of the task class specified by the enclosing <task>. The name and datatype of the slot are specified by the ‘name’ and ‘type’ attributes, respectively.

A slot name shall be a non-colonized XML 1.0 name not containing a dot, hyphen or underscore and not including "external", “when” or “success”, which are predefined slots (see 8.1.2). Two slots within a given <task> element shall not have the same name.

NOTE Dot and hyphen are excluded from slot names so that the rule for constructing corresponding expressions in 9.4 will always result in a valid ECMAScript expression. Dollar sign is already not allowed in XML names.

The input slots of a task class should include all data which affects the execution of task instances. The output slots of a task class should include all data which is modified or created during execution of task instances.

An <input> element shall have a ‘modified’ attribute if and only if the corresponding input data may be modified by the execution of the task. In this case, an output slot of the same datatype as the input slot shall be specified and the value of the 'modified' attribute of the input slot shall be the name of the output slot. The input slot name shall be used in preconditions, postconditions, bindings and constraints to refer to the state of the data object *before* execution; the output slot name shall be used in postconditions, bindings, and constraints to refer to the state of the data *after* execution (modification). See example in 8.6.

NOTE These requirements on the use of the ‘modified’ attribute should not be interpreted as requiring applications (or task engines, more specifically) to copy input data before modifying it. This decision is up to the particular implementation.

8.1.1 Datatypes

The ’type’ attribute specifies a datatype restriction on the values of the slot specified by the pertaining <input> or <output> element. The datatype shall be either one of the primitive ECMAScript [5] datatypes (number, string or boolean) or the ECMAScript identifier of a constructor function. The slot shall contain either a value of the specified type or the undefined value. See 11.7 for application recommendations regarding non-primitive datatypes.

EXAMPLE The following specifies a task class in which the datatype of the slot named “count” is number and the datatype of the slot named “result” is string.

```
<task id="slots">
    <input name="count" type="number"/>
    <output name="result" type="string"/>
</task>
```

8.1.2 Predefined Slots

The following four slots are predefined, which means that every task class has them (even if they are not used). The value of any of these slots may be undefined.

8.1.2.1 Input Slot “device”

If an input slot is named “device” then it shall be of type string and uniquely identify a device. The format of device identification strings is beyond the scope of this standard.

EXAMPLE The value of a task’s “device” slot may be set via a binding in a decomposition class (see 9.4) in order to cause a particular step (e.g., step1) to be executed with a particular household device as the target.

```
<binding slot="$step1.device" value="bedroomTV"/>
```

8.1.2.2 Input Slot “external”

A predefined input slot named “external” of type boolean records whether the task instance is to be or (for task occurrences) was executed by an external entity (value true), or via the grounding interface (see 10) of the application (value false), if known.

NOTE The “external entity” is typically a human being, e.g., the user of the electronic equipment. For example, if the user closes the DVD player door manually, then the value of the “external” slot for the corresponding task occurrence is true. The value is also true if the door is closed by the wind, a power glitch, or the household cat.

EXAMPLE The value of a task’s “external” slot may be tested in the applicability condition of a decomposition class to restrict the use of the decomposition to system tasks.

```
<applicable> ! $this.external </applicable>
```

8.1.2.3 Output Slot "when"

A predefined output slot named "when" of type Date records the moment in time when a task instance should or (for task occurrences) did occur.

8.1.2.4 Output Slot "success"

A predefined output slot named "success" of type boolean records whether a task instance should or (for task occurrences) did succeed (value true) or fail (value false), if known; otherwise the slot is undefined. See 11.6 and 11.9 for application recommendations involving success values.

8.2 User Intent Concepts

DEFINITION 11 A ***user intent concept*** is a case frame, consisting of a verb and a set of semantic roles of specified types.

```
Concept =
    element concept {
        xsd:anyURI |
        ( attribute verb { xsd:QName },
          element role {
              attribute name { xsd:QName },
              # slot shall be the name of task input or output
              # type shall be a primitive ECMAScript type or a constructor
              # defined in the current ECMAScript execution environment
              (
               ( attribute type { DataType }, attribute slot { SlotName }? ) |
               ( attribute slot { SlotName }, attribute type { DataType }? )
              )
          }*
        )
    }
```

Table 4: RelaxNG Schema for User Intent Concepts

The <concept> elements specify the members of the user intent concepts set of the task class specified by the enclosing <task> element, in the order in which the <concept> elements appear. (See 11.4 for application recommendations regarding using the order in which <concept> elements appear.) If there are no <concept> elements, the set is empty.

The <concept> element comes in two forms: it shall either contain a single URI or it shall have a 'verb' attribute and contain zero or more <role> elements. If it contains a URI, then the user intent concept is specified by the URI in a manner which is out of the scope of this standard; otherwise the specified user intent concept is a case frame [10] consisting of the value of the 'verb' attribute and the ordered set of semantic roles [16] specified by the nested <role> elements.

EXAMPLE The following is an example of the URI form of the <concept> element. The URI shown suggests that an OWL [14] ontology is being used to define the user intent concept for myTask.

```
<task id="myTask">
    <concept> http://example.com/concepts.owl#myTask </concept>
</task>
```

Each <role> element specifies a semantic role in the case frame specified by the enclosing <concept> element. The name of the role is specified by the 'name' attribute; the type of the role is either specified directly via the 'type' attribute or indirectly via the 'slot' attribute, which shall be the name of either an input or output slot of the task class specified by the enclosing <task> element. If

the 'type' attribute is omitted, then the type of the specified role is the same as the type of the input or output slot. See 11.4 for application recommendations regarding using the 'slot' attribute of <role> elements.

EXAMPLE The following is an example of the case frame form of the <concept> element. We assume here that the prefix 'av' refers to a namespace in which the verb 'play' and the semantic roles 'audio' and 'video' are defined. The 'theme' and 'location' roles are predefined (see 8.2.1.2 and 8.2.1.4). The non-primitive types Movie, Room, AudioOption and VideoOption are assumed to be defined by scripts elsewhere. An example English sentence expressing this intent concept is "Play Star Wars in the bedroom with Dolby 7.1 and wide screen." Notice that the 'theme' role (type Movie) of the intent concept corresponds directly to the 'content' input slot of the task class, whereas the 'device' input slot (a unique string identifying the networked device) must be computed from the other roles of the intent concept

```
<task id="playMovie">
    <concept verb="av:play">
        <role name="theme" slot="content"/>
        <role name="location" type="Room"/>
        <role name="av:audio" type="AudioOption"/>
        <role name="av:video" type="VideoOption"/>
    </concept>
    <input name="device" type="string"/>
    <input name="content" type="Movie"/>
</task>
```

8.2.1 Predefined Semantic Roles

The following semantic role names are predefined by this standard in the namespace “http://ce.org/cea-2018”.

8.2.1.1 The “agent” Role

Agents are entities that bring about a state of affairs. Furthermore, agents are (or are perceived to be) conscious or sentient, in a way that instruments (8.2.1.3) are not.

EXAMPLE The *italicized entity* in each of the following sentences fills the agent role of the case frame of the underlined verb:

- *The lions* devoured the wildebeest.
- *The boys* caught some fish.
- *My mother* turned on the TV.
- *Suzie* decreased the volume to 5.

8.2.1.2 The “theme” Role

The theme is whatever is acted upon or most affected or undergoes motion of some sort, including motion in a metaphorical sense.

EXAMPLE The *italicized entity* in each of the following sentences fills the theme role of the case frame of the underlined verb:

- Debra broke *the window* with a bat.
- Play *Star Wars* on the TV in the living room.
- Suzie decreased *the volume* to 5.

8.2.1.3 The “instrument” Role

The instrument is whatever is being used to perform the action.

EXAMPLE The *italicized entity* in each of the following sentences fills the instrument role of the case frame of the underlined verb:

- Debra broke the window *with a bat*.
- *This key* opens the door to the main office.

- Play Star Wars *on the TV* in the living room.

8.2.1.4 The "location" Role

Locations are places; they can also serve as the endpoints of paths.

EXAMPLE The *italicized entity* in each of the following sentences fills the location role of the case frame of the underlined verb:

- We put the book *on the shelf*.
- I'd like to send this package *to France*.
- Play Star Wars on the TV *in the living room.*

8.3 ECMAScript Task Instances

A task instance shall be represented in ECMAScript for use in preconditions, postconditions, binding values and grounding as an ECMAScript object with the following properties:

- "model" – a URI identifying the task model in which the task class is defined
- "task" – the task id in the identified task model
- a property corresponding to each (input or output) slot name

The values of the properties corresponding to the task slots shall be the corresponding instance values (including undefined).

8.4 Precondition

DEFINITION 12 A ***precondition*** is a partial boolean function property of a task class with arguments corresponding to the input slots of the task class.

The <precondition> element specifies the precondition function of the task class specified by the enclosing <task> element. The text of the <precondition> element shall be an ECMAScript [5] expression which computes the value of the function and may reference the global variable "$this" containing an instance of the task class. If there is no <precondition> element, the undefined function is the specified precondition.

The content of the <precondition> element shall be properly escaped if containing XML markup characters, or shall be contained in a CDATA section, as specified in [1].

If the precondition of a hypothetical task instance evaluates to false, it is inappropriate to execute the task; otherwise (i.e., if the precondition evaluates to true or undefined) it is appropriate to execute the task if and only if all of its predecessors, if any, in a step ordering (see 9.1) have been successfully executed. Note that even if the precondition is true, this does not guarantee that the task will succeed. See 11.5 for application recommendations regarding preconditions.

EXAMPLE The following specifies a task for loading a DVD, with the precondition that the drawer is open. Notice the $this.device expression returns the value of the "device" input slot of the given task instance.

```
<task id="loadDVD">
    <input name="device" type="string"/>
    <precondition> isDrawerOpen($this.device) </precondition>
</task>
```

NOTE Preconditions can be useful to applications for planning (means-end analysis) or for diagnosing actions inappropriately performed by the user.

8.5 Postcondition

DEFINITION 13 A ***postcondition*** is a partial boolean function property of a task class with arguments corresponding to the input and output slots of the task class.

The <postcondition> element specifies the postcondition function of the task class specified by the enclosing <task> element. The text of the <postcondition> element shall be an ECMAScript [5] expression which computes the value of the function and may reference the global variable “$this” containing an instance of the task class. If there is no <postcondition> element, the undefined function is the specified postcondition.

The content of the <postcondition> element shall be properly escaped if containing XML markup characters, or shall be contained in a CDATA section, as specified in [1].

The specified postcondition is a necessary condition for success of a task. See 8.5.1 for application recommendations regarding postconditions.

8.5.1 The 'sufficient' Attribute

The <postcondition> element has an optional boolean attribute called ‘sufficient’ with default value false.

If the 'sufficient' attribute is true, then the postcondition of the task class specified by the enclosing <task> element is both necessary *and* sufficient for the success of this task. See 11.6 for application recommendations regarding the ‘sufficient’ attribute.

EXAMPLE The task class for opening a (e.g., DVD) drawer below has a sufficient postcondition. We assume here that the global ECMAScript function “isDrawerOpen” is defined elsewhere, e.g., via an initialization script (see 10.1).

```
<task id="openDrawer">
    <input name="device" type="string"/>
    <postcondition sufficient="true"> isDrawerOpen($this.device) </postcondition>
</task>
```

The *only desired effect* of the openDrawer task is to achieve a world state in which the drawer is open, i.e., in which isDrawerOpen($this.device)" returns true. If the drawer is already open, there is no point in trying to perform this task.

EXAMPLE The 'catenate' task specified in 8.6 has a necessary but not sufficient postcondition. If this task is supposed to concatenate two strings, and the length of the output string is not the sum of the two input strings lengths, then clearly the task has failed. On the other hand, just because the output length is correct doesn't mean that the task correctly concatenated the contents of the two input strings.

8.6 Side Effects

The term "side effect" is commonly used in programming languages in two senses. The first sense is when a procedure changes the some aspect of the global state of the computation (e.g., a global variable), which is not declared in the list of inputs to the procedure. In terms of this standard, this is a deficiency in task modeling (see 6).

The second sense of "side effect" is when a procedure modifies a mutable data object which is input to the procedure. In other words, the properties of the data object are different *after* the procedure than before. The optional 'modified' attribute of an input slot provides a way to declare this fact (see 8.1).

In postconditions, the input slot name is used to refer to the data value before modification and the output slot name to the data value after modification.

EXAMPLE Catenate task (only partly specified here) in which the second input string is destructively appended to the first input string:

```
<task id="catenate">
    <input name="string1" type="string" modified="result"/>
    <input name="string2" type="string"/>
```

```
    <output name="result" type="string"/>
    <postcondition> $this.result.length == $this.string1.length + $this.string2.length
</postcondition>
</task>
```

9 TASK DECOMPOSITION

DEFINITION 14 A ***decomposition class*** is a class whose properties include a step order, applicability condition and binding set.

```
Subtasks =
    element subtasks {
        attribute goal { xsd:QName },
        SubtasksContent
    }

SubtasksContent =
    attribute id { xsd:ID },
    (
        ( [a:defaultValue="true"]
            attribute ordered { "true" }?,
            element step {
                StepContent
            }+ )
        |
        ( attribute ordered { "false" },
            element step {
                StepContent,
                # requires shall contain only subtasks step names
                attribute requires { list { StepName+ } }?
            }+ )
    ),
    element applicable {
        # text shall be ECMAScript expression returning boolean or undefined
        # and may reference $this global variable
        text
    }?,
    element binding {
        attribute slot { BindingSlot },
        attribute value {
            # text shall be ECMAScript expression returning a value of same type as
            # the slot corresponding to the 'slot' attribute and may reference $this
            # or global variables corresponding to step names
            text
        }
    }*

BindingSlot =
    # token shall be ECMAScript object property expression starting with $this or
    # a global variable corresponding to a step name, followed by a property
    # corresponding to an input or output slot
    xsd:token { pattern = "$\i[\c-[:.\-]]*.\i[\c-[:.\-]]*" }
```

Table 5: RelaxNG Schema for Subtasks

A <subtasks> element specifies a decomposition class uniquely identified by its ‘id’ attribute, with a predefined slot named “this” (whose value is a task instance) and other slots as specified by the nested <step> elements. The step order property of the decomposition class is specified by the ‘ordered’ attribute of the <subtasks> element and the ‘requires’ attributes of the nested <step> elements, if any (see 9.1) The applicability condition and binding set properties are specified by the nested <applicable> and <binding> elements, respectively (see 9.3 and 9.4).

A <subtasks> element shall be directly nested inside either a <taskModel> element or a <task> element. If it is directly nested inside a <taskModel> element, the task class of the “this” slot of the specified decomposition class shall be the task class specified by its ‘goal’ attribute. If it is directly nested inside a <task> element, it shall not have a ‘goal’ attribute and the task class of the “this” slot shall be the task class specified by the enclosing <task> element.

A task model may contain multiple decomposition classes with the same goal task class.

```
StepContent =
    # two steps within a given subtasks shall not have the same name
    attribute name { StepName },
    attribute task { xsd:QName },
    [ a:defaultValue="1" ]
    attribute minOccurs { xsd:nonNegativeInteger }?,
    [ a:defaultValue="1" ]
    attribute maxOccurs { xsd:positiveInteger | "unbounded" }?

StepName =
    xsd:NCName { pattern = "[^.\-]*" } - "this"
```

Table 6: RelaxNG Schema for Step Content

Every <subtasks> element shall contain at least one <step> element. Each <step> element specifies a task-valued slot of the decomposition class specified by the enclosing <subtasks>. The name and task class of the slot are specified by the ‘name’ and ‘task’ attributes, respectively.

NOTE 1 There is no explicit “include” directive for task model descriptions. Tasks from foreign namespaces can be referenced as goals or steps, simply by using their qualified name, i.e. the concatenation of a namespace prefix, a colon and an ‘id’ value (see example below).

Decompositions may be recursive, i.e., the task class of a step may be the same as the goal class.

NOTE 2 A “step slot” of a decomposition class is different from an “input slot” or “output slot” of a task class. At runtime, a step slot holds a task instance to be carried out as a step for a higher-level task instance.

A step name shall be a non-colonized XML 1.0 name not containing a dot or hyphen and not including "this", which is a predefined slot name. Two steps within a given <subtasks> element shall not have the same name.

NOTE Dot and hyphen are excluded from step names so that the rule for constructing corresponding expressions in 9.4 will always result in a valid ECMAScript expression. Dollar sign is already not allowed in XML names.

All the other elements and attributes in a <subtasks> element are optional.

EXAMPLE The following is a valid, but minimal, decomposition specification. Notice that the step classes of the slots below use a namespace prefix to refer to task classes specified in the example task model in 7.4.

```
<taskModel
        about="http://example.com"
        xmlns="http://ce.org/cea-2018"
        xmlns:my="http://example.com/myproject">
```

```
    <subtasks id="YZ" goal="my:X">
        <step name="step1" task="my:Y"/>
        <step name="step2" task="my:Z"/>
    </subtasks>
</taskModel>
```

NOTE A decomposition classes can be thought of as a “rule” for decomposing a given instance of the goal task type. The result of “applying” the rule is an instance of the decomposition class in which the given instance is the value of the “this” slot and each step slot contains an instance (typically newly constructed) of the corresponding task class.

EXAMPLE Applying the example decomposition class YZ above to a given instance of X would result in an instance of YZ containing the given instance of X as the value of the “this” slot, an instance of Y in the "step1" slot and an instance of Z in the "step2" slot.

9.1 Step Order

The step order property of a decomposition class shall be a partial order on the step names. See 11.3 for application recommendations regarding step order.

If the ‘ordered’ attribute of a <subtasks> element is true (the default), the steps are totally ordered in the sequence in which the <step> elements appear, and the <step> elements shall not have a ‘requires’ attribute. If the 'ordered' attribute is false, the steps are unordered except as restricted by the 'requires' attributes of the <step> elements.

The 'requires' attribute of a <step> element shall be a whitespace-separated list of the names of other steps specified in the same <subtasks> element. Each listed step specifies an ordering restriction in which the listed step precedes the step which has the ‘requires’ attribute.

NOTE A partial order does not allow circularities in ordering restriction.

EXAMPLE The following specifies a partial order in which step3 must occur after step1 and step2, but step1 and step2 may occur in any order.

```
<subtasks id="DEF" goal="my:X" ordered="false">
    <step name="step1" task="D"/>
    <step name="step2" task="E"/>
    <step name="step3" task="F" requires="step1 step2"/>
</subtasks>
```

NOTE The notion of partial order used should not be confused with multi-threaded parallelism. If two steps are unordered, that does *not* mean that they can occur in parallel; it means they can occur in *either* order. A partial order is simply a compact way of representing a set of allowable sequences. For example, the DEF decomposition above specifies that both and only the sequences [step1, step2, step3] and [step2, step1, step3] are allowed. A partial order is the mathematical construct commonly used in planning systems for this purpose.

9.2 Skipping and Repeating Steps

In the semantics of decomposition classes, the optionality and repeatability of steps is represented via two integer-valued properties, called minimum occurrences and maximum occurrences, which are specified by the optional ‘minOccurs’ and ‘maxOccurs’ attributes of the <step> element, respectively. The default value for both of these attributes is ”1”, which means that the specified step is not intended to either be skipped or repeated. See 11.5 for application recommendations regarding skipping and repeating steps.

A ‘maxOccurs’ value of “unbounded” specifies that there is no upper limit on the number of step repetitions.

Notice that repeating a step does *not* mean adding an additional slot; repeating a step means replacing the current value of a step slot with a new task instance.

Loops with multiple steps in the body of the loop can be specified using the common programming technique of tail recursion, wherein the type of the last step of a decomposition is the same as the goal.

EXAMPLE The decomposition class below encapsulates the strategy of repeatedly power cycling a device (e.g., a DVD player) until you succeed in opening the drawer. Notice that the first step of the decomposition refers to the openDrawer task defined in 8.5.1, with maxOccurs set to 1 to prevent immediate retrying if it fails (see 11.9). The <binding> elements (see 9.4) guarantee that all the steps are applied to the same device.

```
<task id="openDrawerHarder">
    <input name="device" type="string"/>
    <postcondition sufficient="true"> isDrawerOpen($this.device) </postcondition>
    <subtasks id="harder">
        <step name="open" task="openDrawer" maxOccurs="1"/>
        <step name="off" task="powerOff"/>
        <step name="on" task="powerOn"/>
        <step name="again" task="openDrawerHarder"/>
        <binding slot="$open.device" value="$this.device"/>
        <binding slot="$off.device" value="$this.device"/>
        <binding slot="$on.device" value="$this.device"/>
        <binding slot="$again.device" value="$this.device"/>
    </subtasks>
</task>
```

9.3 Applicability Condition

DEFINITION 15 An ***applicability condition*** is a partial boolean function property of a decomposition class or a script with arguments corresponding to the input slots of the goal or script task class.

The <applicable> element specifies the applicability condition of the decomposition class specified by the enclosing <subtasks> element. The text of the <applicable> element shall be an ECMAScript [5] expression which computes the value of the function and may reference the global variable “$this” containing an instance of the goal task class. If there is no <applicable> element, the undefined function is the specified applicability condition. See 10.1 for the specification of the applicability condition of a script.

The content of the <applicable> element shall be properly escaped if containing XML markup characters, or shall be contained in a CDATA section, as specified in [1].

If the applicability condition of a decomposition class evaluates to false for the arguments corresponding to the input slots of a given instance of the goal task class, it is inappropriate to use this decomposition class to decompose that goal instance. Note that even if the applicability condition is true, this does not guarantee that executing the steps of the decomposition will lead to the success of the goal task. See 11.5 for application recommendations regarding the applicability condition.

NOTE Applicability conditions are typically used to distinguish between alternative task decompositions for the same goal class. Zero, one or more applicability conditions of decompositions for the same goal class may be true at the same time.

NOTE The applicability condition is only intended to be evaluated immediately before execution of subtasks commences. It is no longer relevant after the first subtask has been started. For this reason, the applicability condition is not used in task recognition.

9.4 Bindings

DEFINITION 16 A ***binding set*** is a (possibly empty) set property of a decomposition class containing bindings.

DEFINITION 17 A ***binding*** is an equality between an input slot of a decomposition step or an output slot of a decomposition goal, and the value of a function with arguments corresponding either to the output slots of steps or the input slot of the goal.

The <binding> elements specify the binding set of the decomposition class specified by the enclosing <subtasks> element. The members of the binding set are the bindings specified by each <binding> element. If there are no <binding> elements, the binding set is empty.

Each <binding> element specifies a binding between the slot corresponding to the value of its 'slot' attribute and the function computed by the ECMAScript expression that is its 'value' attribute.

The 'slot' attribute of a <binding> element shall contain an ECMAScript expression starting with "$this" or a global variable constructed by prefixing dollar sign to a step name specified by the enclosing <subtasks> element, followed by a property corresponding to an output slot of the goal or the input slot of the step.

NOTE In a future version of this standard, the 'slot' attribute may be generalized to allow an arbitrary expression. This extension would give the binding set the expressive power of a constraint solving system.

The 'value' attribute shall be an ECMAScript expression returning a value of the same type as the slot corresponding to the 'slot' attribute. This expression may reference "$this" (containing an instance of the goal type) or global variables constructed by prefixing dollar sign to a step name specified by the enclosing <subtasks> element (containing instances of the corresponding step type).

The content of the 'value' attribute shall be properly escaped if containing XML markup characters, or shall be contained in a CDATA section, as specified in [1].

The specified binding set, viewed as a directed graph, shall be acyclic. A binding set is viewed as a directed graph as follows: Each variable that appears anywhere in a binding corresponds to a node in the graph; each appearance of a slot in the 'value' part of a binding corresponds to an arc in the graph directed from the node corresponding to that slot to the node corresponding to the slot in the 'slot' part of the binding.

NOTE The binding set can be thought of as specifying the "data flow" between steps in a decomposition. In task generation, the bindings are used to compute the inputs of the next step to be performed, e.g., by setting the slot corresponding to the 'slot' attribute to the result of evaluating the 'value' expression. In task recognition, they restrict the inference to those patterns of task instances compatible with the bindings.

EXAMPLE In the most common usage of the <binding> element, the 'value' expression is a simple reference to a property of one of the global variables. This gives rise to the three basic binding patterns (input-input, input-output and output-output) shown below and illustrated in Figure 5 using the traditional data flow arrow notation.

```
<task id="A">
    <input name="input1" type="number"/>
    <input name="input2" type="string"/>
    <output name="output1" type="number"/>
    <output name="output2" type="string"/>
</task>

<task id="B">
    <input name="input" type="string"/>
    <output name="output1" type="number"/>
    <output name="output2" type="string"/>
</task>

<task id="C">
    <input name="input1" type="number"/>
```

```
    <input name="input2" type="string"/>
    <output name="output" type="number"/>
</task>

<subtasks id="BC" goal="A">
    <step name="b" task="B"/>
    <step name="c" task="C"/>
    <binding slot="$c.input1" value="$this.input1"/>
    <binding slot="$b.input" value="$this.input2"/>
    <binding slot="$c.input2" value="$b.output1"/>
    <binding slot="$this.output1" value="$c.output"/>
    <binding slot="$this.output2" value="$b.output2"/>
</subtasks>
```

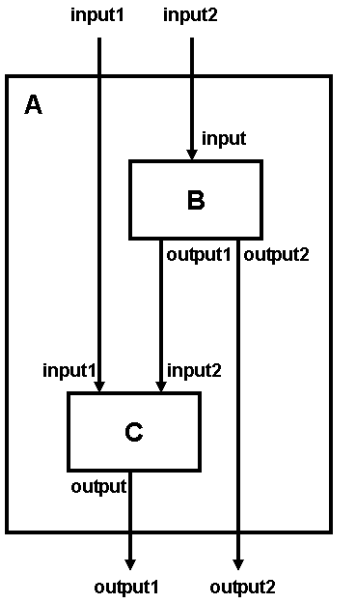

Figure 5: Dataflow Diagram for BC Decomposition

10 GROUNDING

In order for task models to be useful, applications need to connect (*ground*) aspects of a task model to the physical world of users and devices. Grounding is a bidirectional connection: applications need to execute tasks to change the state of the world (see 10.1) and query the state of the physical world (see 10.2); applications also need to be notified when the state of the world changes due to external agencies, such as the user (see 11.10.3). Furthermore, the implementation of grounding typically involves the use of one or more specific networking platforms.

10.1 Scripts

DEFINITION 18 A ***script*** is an ECMAScript [5] program which may be associated with one or more tasks classes, platforms and device types and whose properties include an applicability condition.

```
Script =
    element script {
        ( ( attribute task { xsd:QName }+,
            attribute model { xsd:anyURI }+ ) |
          [ a:defaultValue="false" ]
          attribute init { xsd:boolean }?
        ),
        ScriptContent
    }

ScriptContent =
    attribute platform { token }?,
    attribute deviceType { token }?,
    # text shall be ECMAScript expression returning boolean or undefined
    # and unless init may reference $this global variable
    attribute applicable { text }?,
    # text shall be ECMAScript program and unless init is true
    # may reference $this, $platform and $deviceType global variables
    text
```

Table 7: RelaxNG Schema for Scripts

Platform and device specific execution grounding of tasks is achieved by delegating to scripts associated with the appropriate task class. The <script> element specifies a script with optional task class, platform and device type associations specified by its 'task', 'model', 'platform' and 'deviceType' attributes, and applicability condition specified by its optional 'applicable' property. A <script> element shall be directly nested inside either a <taskModel> element or a <task> element. If it is directly nested inside a <task> element, it shall not have a 'task' or 'model' attribute.

Each 'task' attribute identifies a task class with which the specified script is associated. Each 'model' attribute specifies that the script is associated with all of the task classes in the identified task model. If there are no 'task' or 'model' attributes, then the specified script is associated with *all* task classes which do not otherwise have associated scripts, unless the 'init' attribute is true.

NOTE 1 Scripts are typically associated with primitive task classes, i.e., classes which have no applicable decomposition classes. However, in general, a script may be associated with any task class. It is up to the application to decide between using a decomposition class or a script, if there is a choice.

NOTE 2 A script associated with all tasks classes (in a model) can be convenient, for example, to ground all tasks for a given platform via a single generic procedure. See 11.10.1 for more details.

DEFINITION 19 An ***initialization script*** is a script which is not associated with any task class and is intended to be executed exactly once.

If the 'init' attribute is true, then the specified script is an initialization script. Initialization scripts may define functions, constructors, etc., used in other scripts and in datatype restrictions. There may be more than one initialization script.

The 'platform' attribute identifies the network platform with which the specified script is associated. If it is omitted, the specified script is associated with *all* platforms. The format of platform identification strings is outside the scope of this standard.

The 'deviceType' attribute identifies the device type with which the specified script is associated. If it is omitted, the script is associated with *all* device types. The format of device type identification strings is outside the scope of this standard.

The 'applicable' attribute specifies the applicability condition of the specified script. The content of the 'applicable' element shall be an ECMAScript [5] expression which computes the value of the

applicability function and may reference the global variable “$this” containing an instance of the associated task class. If there is no ‘applicable’ attribute, the undefined function is the specified applicability condition.

NOTE 3 The applicability condition is an attribute here rather than an element, as in decomposition classes (see 9.3), because the <script> element already uses text content for its ECMAScript program.

The content of the ‘applicable’ attribute shall be properly escaped if containing XML markup characters, or shall be contained in a CDATA section, as specified in [1].

If the applicability condition of a script evaluates to false for a given instance of the task class, it is inappropriate to use this script.

The text of a <script> element shall be an ECMAScript [5] program and (except for initialization scripts) may reference the global variable “$this” (containing an instance of the associated task class) in order to read input slot values and set output slot values, including the “success” slot. See 11.10 for application recommendations regarding execution of scripts.

The text content of the <script> element shall be properly escaped if containing XML markup characters, or shall be contained in a section, as specified in [1].

EXAMPLE 1 If not used as markup, the characters ‘<’, ‘>’ and ‘&’ are XML encoded as ‘<’, ‘>’ and ‘&’, respectively. Thus the following script contains an escaped ‘>’ character:

```
<script> if  (i &lt; max) i = i + 1; </script>
```

EXAMPLE 2 The same code contained in a CDATA section:

```
<script> <![CDATA[ if  (i < max) i = i + 1; ]]> </script>
```

10.2 Grounding Queries

Table 8 specifies a predefined task model for grounding queries that is part of this standard. Applications can use the platform and device specific scripts associated with the task classes in this model to appropriately query the state of the world. The first two task classes in Table 8 are general queries; the last two query a specific device.

NOTE Scripts can easily be associated with these task classes by including the scripts in other task models and referencing these task classes using fully qualified names in the ‘task’ attribute.

```
<taskModel
        about="http://ce.org/cea-2018/query"
        xmlns="http://ce.org/cea-2018">

    <task id="devices">
        <output name="devices" type="Array"/>
    </task>

    <task id="user">
        <output name="user" type="string"/>
    </task>

    <task id="about">
        <input name="device" type="string"/>
        <output name="about" type="string"/>
    </task>

    <task id="task">
        <input name="device" type="string"/>
```

```
        <input name="task" type="string"/>
        <output name="model" type="string"/>
    </task>

</taskModel>
```

Table 8: Predefined Task Model for Grounding Queries

10.2.1 The 'devices' Task

The 'devices' task class returns a (possibly zero length) array of strings identifying the devices currently known to the system. The format of these strings is beyond the scope of this standard, but it shall be consistent with the format used by the predefined 'device' input slot (see 8.1.2.1). See 11.11 for an application recommendation regarding the 'devices' task.

10.2.2 The 'user' Task

The 'user' task class returns a string indicating which human user is currently interacting with the application or undefined if there is no user. The format of the string is beyond the scope of this standard.

10.2.3 The 'about' Task

The 'about' task class returns an XML document describing the device identified by the 'device' input and conforming to the syntax given in Table 9, with additional attributes allowed.

```
default namespace = "http://ce.org/cea-2018/about"
namespace xsd = "http://www.w3.org/2001/XMLSchema"

start =
    element about {
        attribute device { token },
        attribute type { token },
        attribute friendlyName { token },
        attribute manufacturer { xsd:anyURI },
        attribute friendlyManufacturer { token },
        attribute model { token },
        attribute serial { token },
        attribute taskModel { xsd:anyURI }?
    }
```

Table 9: RelaxNG Schema for About Document

The value of the 'device' attribute in the root element of this document shall be the same as the input device string. The value of the optional 'taskModel' attribute shall be a URI from which a task model description conforming to this standard may be obtained.

> NOTE In UPnP or other IP-based network platforms, the task model URI can point back to the device itself, so that the device can effectively upload its own task model.

The format of the other attributes is beyond the scope of this standard. However, the format of the 'type' attribute shall be consistent with the format used by the 'deviceType' attribute of the <script> element (see 10.1). See Annex C and D for recommendations regarding the correspondence between attributes of the <about> element and platform-specific device self-descriptions.

10.2.4 The 'task' Task

The 'task' input to this task shall be the fully qualified name of a task class. The output of this task shall be a URI from which a task model description conforming to this standard may be obtained contained containing the specification of the given task class.

> NOTE This task is included to support situations, e.g., in web services, where there is a very large set of possible task classes.

11 RECOMMENDATIONS FOR APPLICATIONS

This section contains recommendations for the behavior of applications with respect to a given task model (see Figure 1). It is strongly suggested that applications adhere to these recommendations.

11.1 Unknown Elements and Attributes

In order to facilitate flexibility and extensions to this standard, applications should ignore (or at least process without error) unknown elements and attributes, including elements and attributes from unknown or unsupported namespaces.

> EXAMPLE For example, task model authors may wish to add elements from the namespaces of the Dublin Core Metadata Initiative [8][9] (see example in 6.7).

11.2 Task Model Ordering

Applications should in general use the order of definition of elements within a task model as the default and preference orders. For example, the definition order should be the default order for presenting task choices to the user and the preference order in which to try alternative decompositions or scripts for a given task. If multiple copies of the same task model are loaded, the application should use the definitions in the most recently loaded version.

11.3 Order of Step Execution

Applications should execute the steps in a task decomposition in a sequence which is consistent with the step order (see 9.1). Furthermore, the subtasks (if any) of two steps should not be interleaved.

> EXAMPLE In the DEF decomposition example in 9.1, if step1 and step2 are each decomposed into two subtasks, the two subtasks of step1 should be executed before the two subtasks of step2, or vice versa.

11.4 User Intent Concepts

Applications should use the 'slot' attribute in <role> elements as an indication of how to map the semantic role values of an instance of a user intent concept to the slots of a task instance.

Applications should use the order of elements in the intent concepts set as the preference order for resolving user intent.

Applications should use the order of roles in a case frame as the default order for presenting choices to the user.

11.5 Preconditions

Applications should not attempt to execute a task whose precondition (see 8.4) evaluates to false.

11.6 Postconditions

Applications should not attempt to execute a task whose sufficient postcondition (see 8.5.1) is true.

The remainder of the recommendations in this subsection address constraints between the 'success' slot of a task occurrence and its postcondition:

- If the value of the ‘success’ slot (see 8.1.2.4) of a task occurrence is undefined and the postcondition (see 8.5) of the task is true or false, applications should conclude that the task succeeded or failed, respectively.
- If the value of the ‘success’ slot of a task occurrence is true and the postcondition of the task is undefined, applications should conclude that the postcondition is true.
- If the value of the ‘success’ slot of a task occurrence is false and the sufficient postcondition of the task is undefined, applications should conclude that the postcondition is false.
- If the value of the ‘success’ slot of a task occurrence is true and the postcondition of the task is false, or the value of the ‘success’ slot is false and the sufficient postcondition is true, then applications should conclude there is a modeling error, i.e., the task model is inconsistent with the “real world”.

11.7 Non-primitive Datatypes

Applications should provide an ECMAScript [5] execution environment in which constructor functions are defined corresponding to all non-primitive datatypes (see 8.1.1) used in loaded task models.

NOTE A typical method of achieving this is to define or load the needed pseudoclasses in the initialization scripts of the task models.

11.8 Applicability Conditions

Applications should not apply decompositions (see 9) or execute scripts (see 11.10) whose applicability condition is false before decomposition or execution has commenced.

11.9 Skipping and Repeating Steps

Applications should not execute a step fewer than the number of times given by the ‘minOccurs’ attribute or more times than the value of the ‘maxOccurs’ attribute. A step should not be repeated after any other step has been executed. Applications may automatically repeat a step if it failed, if the ‘maxOccurs’ value allows.

11.10 Script Execution

Applications should execute each initialization script exactly once, e.g., when the containing task model is loaded. The initialization scripts containing in a given task model should be executed in the order defined and without intervening execution of other scripts.

Before executing a script, applications should first initialize the following global variables in the ECMAScript execution environment:

- $this – an ECMAScript representation (see 8.3) of a hypothetical instance of an associated task class
- $platform -- a string identifying a platform (consistent with 'platform' attribute of <script> elements)
- $deviceType -- a string identifying a device type (consistent with 'deviceType' attribute of <script> elements), e.g., the type of the value of the ‘device’ input slot, if any

11.10.1 Predefined ‘$execute’ Function

The application's ECMAScript [5] execution environment should contain an ECMAScript function named '$execute' with the two arguments:

- a string identifying a platform (consistent with 'platform' attribute of <script> elements)
- a string identifying a device type (consistent with 'deviceType' attribute of <script> elements)

This function should cause execution of the hypothetical task instance represented by the current value of the “$this” global variable. The results (outputs) of the execution should be reflected in the properties of the “$this” object when the call to '$execute' returns.

NOTE 1 It is the script author’s responsibility to save and restore the current value of “$this”.

If there is an appropriate script associated with task instance, platform and device type, an application may implement the ‘$execute’ function by executing this script as described above. However, the application may alternatively use appropriate decomposition classes, if available, before then grounding the lower level task classes using scripts.

NOTE 2 The ‘$execute’ function thus supports a limited kind of interleaving between hierarchical decomposition and decomposition by scripts.

11.10.2 Predefined ‘$getModel’ Function

The application's ECMAScript [5] execution environment should contain an ECMAScript function named '$getModel’ with one string argument identifying a task model by its URI. The function should return a Document Object Model for the corresponding task model description.

11.10.3 Predefined ‘$occurred’ Function

The application's ECMAScript [5] execution environment should contain an ECMAScript function named '$occurred’ with one argument that is expected to be the ECMAScript representation of a task occurrence (see 8.3). The function should inform the application that an external event occurred.

11.11 Grounding Queries

An application should use the order of devices returned by the ‘devices’ task as the default order for presenting devices to the user.

Annex A
(Normative)
Complete RelaxNG Schema for Task Model Description

A task model description shall comply with the following syntax, specified in RelaxNG compact notation [6].

NOTE This appendix reproduces the contents of Tables 1 through 7 in this standard.

```
default namespace = "http://ce.org/cea-2018"

namespace xsd = "http://www.w3.org/2001/XMLSchema"
namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
namespace dc = "http://purl.org/dc/elements/1.1"
namespace dcterms = "http://purl.org/dc/terms"

start =
    element taskModel {
        # about shall be non-empty and not contain fragment identifier
        attribute about { xsd:anyURI },
        Description*,
        ( Task | Subtasks | Script )*
    }

Description =
    element dc:* { attribute * { text }*, text? } |
    element dcterms:* { attribute * { text }*, text? }

Task =
    element task {
        attribute id { xsd:ID },
        Description*,
        Concept*,
        Input*,
        Output*,
        element precondition {
            # text shall be ECMAScript expression returning boolean or undefined
            # and may reference $this global variable
            text
        }?,
        element postcondition {
            [ a:defaultValue="false" ]
            attribute sufficient { xsd:boolean }?,
            # text shall be ECMAScript expression returning boolean or undefined
            # and may reference $this global variable
            text
        }?,
        element subtasks { SubtasksContent }*,
        element script { ScriptContent }*
    }

Concept =
```

```
    element concept {
        xsd:anyURI |
        ( attribute verb { xsd:QName },
          element role {
              attribute name { xsd:QName },
              # slot shall be the name of task input or output
              # type shall be a primitive ECMAScript type or a constructor
              # defined in the current ECMAScript execution environment
              (
               ( attribute type { DataType }, attribute slot { SlotName }? ) |
               ( attribute slot { SlotName }, attribute type { DataType }? )
              )
          }*
        )
    }

Input =
    element input {
        Slot,
        # modified shall be the name of an output slot
        attribute modified { SlotName }?
    }

Output =
    element output { Slot }

Slot =
    # two slots within a given task shall not have the same name
    attribute name { SlotName },
    # type shall be primitive ECMAScript datatype or constructor defined
    # in current ECMAScript execution environment
    attribute type { DataType }

SlotName =
    xsd:NCName { pattern = "[^.\-]*" } - ("external" | "when" | "success")

DataType =
    "number" | "string" | "boolean" | xsd:token { pattern = "(\i|$)([\c-[:.\-]]|$)*" }


Subtasks =
    element subtasks {
        attribute goal { xsd:QName },
        SubtasksContent
    }

SubtasksContent =
    attribute id { xsd:ID },
    (
        ( [a:defaultValue="true"]
            attribute ordered { "true" }?,
            element step {
                StepContent
            }+ )
        |
        ( attribute ordered { "false" },
```

```
            element step {
                StepContent,
                # requires shall contain only subtasks step names
                attribute requires { list { StepName+ } }?
            }+ )
    ),
    element applicable {
        # text shall be ECMAScript expression returning boolean or undefined
        # and may reference $this global variable
        text
    }?,
    element binding {
        attribute slot { BindingSlot },
        attribute value {
            # text shall be ECMAScript expression returning a value of same type as
            # the slot corresponding to the 'slot' attribute and may reference $this
            # or global variables corresponding to step names
            text
        }
    }*

BindingSlot =
    # token shall be ECMAScript object property expression starting with $this or
    # a global variable corresponding to a step name, followed by a property
    # corresponding to an input or output slot
    xsd:token { pattern = "$\i[\c-[:.\-]]*.\i[\c-[:.\-]]*" }

StepContent =
    # two steps within a given subtasks shall not have the same name
    attribute name { StepName },
    attribute task { xsd:QName },
    [ a:defaultValue="1" ]
    attribute minOccurs { xsd:nonNegativeInteger }?,
    [ a:defaultValue="1" ]
    attribute maxOccurs { xsd:positiveInteger | "unbounded" }?

StepName =
    xsd:NCName { pattern = "[^.\-]*" } - "this"


Script =
    element script {
        ( ( attribute task { xsd:QName }+,
            attribute model { xsd:anyURI }+ ) |
          [ a:defaultValue="false" ]
          attribute init { xsd:boolean }?
        ),
        ScriptContent
    }

ScriptContent =
    attribute platform { token }?,
    attribute deviceType { token }?,
    # text shall be ECMAScript expression returning boolean or undefined
    # and unless init may reference $this global variable
    attribute applicable { text }?,
```

```
# text shall be ECMAScript program and unless init is true
# may reference $this, $platform and $deviceType global variables
text
```

Annex B
(Informative)
Example Task Model

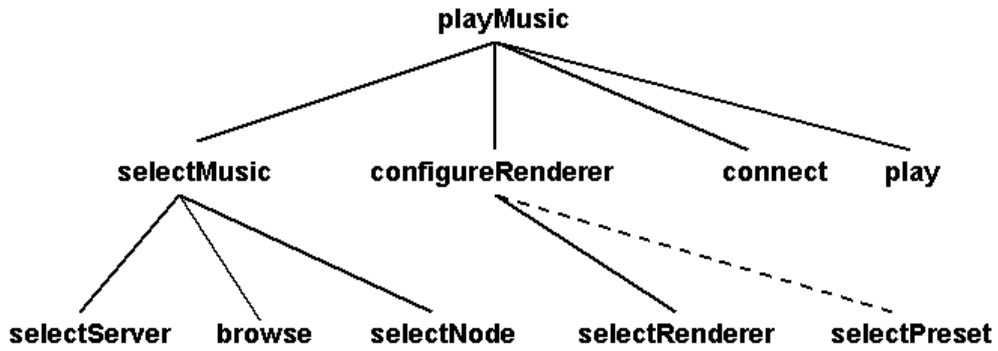

Figure 6: Decomposition for playMusic Task

This annex contains below the XML source for a complete CEA-2018 task model description in the domain of audio-visual devices, illustrating many of the features of standard. The high-level organization of the tasks in the model is based on the UPnP AV architecture [17]. The grounding is based on the URC [11] socket abstraction of the UPnP AV architecture.

This task model contains a single toplevel task, playMusic, which is decomposed as shown in Figure 6. The dotted line in Figure 6 indicates an optional step. Notice in the task model description below that the four steps in the decomposition of playMusic are only partially ordered: selectMusic and configureRenderer must occur (in either order) before connect, which then must be followed by play. Other steps are totally ordered.

The seven primitive tasks in Figure 6 (at the leaves of the tree) are designed to correspond to variables and commands in the URC socket abstraction of the UPnP AV architecture. See the comments in the script stubs below for more details of how the URC grounding operates. To ground this task model directly in UPnP, either further decompositions could be provided or more elaborate individual scripts for each of these primitive tasks.

Notice that the five primitive tasks defined at the end of the task model description below do not appear in the tree in Figure 6. These tasks are used only for abstracting error notifications, which are external events and therefore not in the planned decomposition for playMusic. The application may have other task models which specify how to respond to such events.

Notice also that some step input slots, such as $connect.preferredConnectionPrototocol in playMusicSteps below, are bound to values by the decomposition class, while others, such as $select.selectedMediaServer (in the same decomposition class), are not. In general, it is the responsibility of the application to obtain values for unbound input slots, typically by asking the user. The details of the user interface by which this information is obtained from the user is out of the scope of this standard.

Finally, notice that since URC sockets preserve state information in variables, some dataflow can be omitted between steps in this model. For example, once the value of the selectedMediaServer variable is set from the input slot of selectServer, it is used for the rest of the execution. A dataflow binding is required, however, between the $connect.newConnectionId and $play.connectionId in playMusicSteps, because the URC socket does not keep track of which of several open connections should be played next.

```
<taskModel
        about="http://ce.org/cea-2018/AnnexB"
        xmlns="http://ce.org/cea-2018"
        xmlns:dc="http://purl.org/dc/elements/1.1"
        xmlns:dcterms="http://purl.org/dc/terms">

    <dc:title xml:lang="en">Playing Music</dc:title>
    <dc:description xml:lang="en">CEA-2018 conformant sample task model description for
        playing music with UPnP AV devices and URC grounding.</dc:description>
    <dc:creator>Gottfried Zimmermann</dc:creator>
    <dc:contributor>Charles Rich</dc:contributor>
    <dcterms:issued>2007-08-25</dcterms:issued>
    <dcterms:modified>2007-09-10</dcterms:modified>

    <task id="playMusic">
        <subtasks id="playMusicSteps" ordered="false">
            <step name="select" task="selectMusic"/>
            <step name="configure" task="configureRenderer"/>
            <step name="connect" task="connect" requires="select configure"/>
            <step name="play" task="play" requires="connect"/>
            <binding slot="$connect.preferredConnectionProtocol" value="'*'"/>
        </subtasks>
    </task>

    <task id="selectMusic">
        <subtasks id="selectMusicSteps">
            <step name="server" task="selectServer"/>
            <step name="browse" task="browse"/>
            <step name="node" task="selectNode"/>
            <binding slot="$browse.browseFilter" value="'*'"/>
            <binding slot="$browse.browseSortCriteria" value="'+dc:title'"/>
            <binding slot="$play.connectionId" value="$connect.newConnectionId"/>
        </subtasks>
    </task>

    <task id="configureRenderer">
        <subtasks id="configureRendererSteps">
            <step name="select" task="selectRenderer"/>
            <step name="preset" task="selectPreset" minOccurs="0" maxOccurs="1"/>
        </subtasks>
    </task>

    <!-- primitive tasks and grounding -->

    <script platform="URC">
        // Default grounding script goes here, which executes primitive tasks below by:
        // (1) setting socket variables from values of same-named $this input slots
        // (2) invoking same-named URC command (if any)
        // (3) setting $this output slots from same-named socket variables
        // (4) setting $this.success to reflect success or failure of command (if any)
    </script>

    <script platform="URC" init="true">
        // Initialization script goes here to:
        // (1) define PlayMode, PlaySpeed and ErrorDescription types used below
        // (2) install URC notification listener which creates same-named external task
        //     occurrence below and invokes $occurred
    </script>

    <task id="selectServer">
        <input name="selectedMediaServer" type="string"/>
    </task>
```

```
    <task id="browse">
        <input name="browseFilter" type="string"/>
        <input name="browseSortCriteria" type="string"/>
    </task>

    <task id="selectNode">
        <input name="selectedNodeId" type="string"/>
    </task>

    <task id="selectRenderer">
        <input name="selectedMediaRenderer" type="string"/>
    </task>

    <task id="selectPreset">
        <input name="presetName" type="string"/>
    </task>

    <task id="connect">
        <input name="preferredConnectionProtocol" type="string"/>
        <output name="newConnectionId" type="string"/>
        <output name="error" type="ErrorDescription"/>
    </task>

    <task id="play">
        <input name="connectionId" type="string"/>
        <input name="playCurrentPlayMode" type="PlayMode"/>
        <input name="playTransportPlaySpeed" type="PlaySpeed"/>
    </task>

    <!-- external events -->

    <task id="transportStatusError">
        <output name="error" type="ErrorDescription"/>
    </task>

    <task id="conNotifyContentFormatMismatch">
        <output name="error" type="ErrorDescription"/>
    </task>

    <task id="conNotifyInsufficientNetworkResources">
        <output name="error" type="ErrorDescription"/>
    </task>

    <task id="conNotifyUnreliableChannel">
        <output name="error" type="ErrorDescription"/>
    </task>

    <task id="conNotifyUnknownConnectionError">
        <output name="error" type="ErrorDescription"/>
    </task>

</taskModel>
```

Annex C
(Normative)
UPnP Devices Exposing a Task Model Description

A UPnP device conforming to this standard may provide a reference to a conforming task model description through its UPnP Device Description [17] as follows:

The UPnP Device Description of the device shall include a <taskModel> element of namespace http://ce.org/cea-2018/upnp-device-description-ext as subelement of <device>. A URI [3] shall be provided as element content, referencing a conforming task model description for the device. The URI shall be either locally or globally resolvable, i.e., any other device in the local network can retrieve a task model description through this URI.

EXAMPLE 1 The following device description fragment is used by a UPnP device to point to a task model description served by its built-in web server:

```
<cea-2018:taskModel
    xmlns:cea-2018="http://ce.org/cea-2018/upnp-device-description-ext">
    http://192.168.0.10/cea-2018/taskmodel.xml
</cea-2018:taskModel>
```

EXAMPLE 2 The following device description fragment is used by a UPnP device to point to a task model description served by the web server of its manufacturer:

```
<cea-2018:taskModel
    xmlns:cea-2018="http://ce.org/cea-2018/upnp-device-description-ext">
    http://example.com/modelX/taskmodel.xml
</cea-2018:taskModel>
```

When serving a task model description from a URI, the web server should indicate its MIME type as specified in 7.1.

When building an 'about' document (see 10.2.3) for a UPnP device, the following correspondences should be used:

Attribute in CEA-2018 'about' document	**Element in UPnP Device Description**
device	UDN
deviceType	deviceType
friendlyName	friendlyName
manufacturer	manufacturerURL
friendlyManufacturer	manufacturer
model	modelDescription
serial	serialNumber
taskModel	cea-2018:taskModel (see above specification)

Annex D (Normative)
URC Targets Exposing a Task Model Description

A URC target device conforming to this standard may provide a reference to a conforming task model description through its target description [12] as follows:

The target description of the device shall include a <taskModel> element of namespace http://ce.org/cea-2018/target-description-ext as subelement of <target>. A URI [3] shall be provided as element content, referencing a conforming task model description for the device. The URI provided shall be either locally or globally resolvable, i.e., any other device in the local network can retrieve a task model description through this URI. If a relative URI is given, its base is determined by the location of the target description.

If a task model pertains to a single socket of a URC target only, the <taskModel> element should be a subelement of the pertaining <socket> element.

EXAMPLE 1 The following target description fragment is used by a URC target device to point to a task model description that spans multiple of its sockets (ellipses used to mark skipped content):

```
<target
    about="http://example.com/thermometer"
    id="target"
    xmlns="http://myurc.org/ns/targetdesc">
    ...
    <cea-2018:taskModel
        xmlns:cea-2018=”http://ce.org/cea-2018/target-description-ext”>
        http://example.com/thermometer/taskmodel.xml
    </cea-2018:taskModel>
    ...
</target>
```

EXAMPLE 2 The following target description fragment is used by a URC target device to point to a task model description that pertains to one of its sockets (ellipses used to mark skipped content):

```
<target
    about="http://example.com/thermometer"
    id="target"
    xmlns="http://myurc.org/ns/targetdesc">
    ...
    <socket id="socket1" name="http://example.com/thermometer/socket1">
        <socketDescriptionLocalAt>socket1/socketdescription.xml</socketDescriptionLocalAt>
        <cea-2018:taskModel
            xmlns:cea-2018=”http://ce.org/cea-2018/target-description-ext”>
            socket1/taskModel.xml       <!-- relative to target description -->
        </cea-2018:taskModel>
    </socket>
    <socket id="socket2" name="http://example.com/thermometer/socket2">
        <socketDescriptionLocalAt>socket2/socketdescription.xml</socketDescriptionLocalAt>
    </socket>
    ...
</target>
```

When serving a task model description from a URI, the web server should indicate its MIME type as specified in 7.1.

When building an ‘about’ document (see 10.2.3) for a URC target device, the following correspondences should be used:

Attribute in CEA-2018 'about' document	**Information Item in URC Framework**
device	target instance identifier [11]
deviceType	'about' attribute of <target> in target description
friendlyName	label (resource) for target instance identifier if existing; otherwise label for 'about' attribute of <target> element
manufacturer	<dc:creator> (subelement of <target>) in target description
friendlyManufacturer	label (resource) for <dc:creator> element if existing; otherwise <dc:creator> element content
model	<dc:title> in target description
serial	<dc:identifier> in target description
taskModel	cea-2018:taskModel (see above specification)

CEA Document Improvement Proposal

If in the review or use of this document, a potential change is made evident for safety, health or technical reasons, please email your reason/rationale for the recommended change to standards@ce.org.

Consumer Electronics Association
Technology & Standards Department
1919 S Eads Street, Arlington, VA 22202
FAX: (703) 907-7693 standards@CE.org


CEA
Consumer Electronics Association