Informe

Guillermo Pacheco

25 de julio de 2014

Capítulo 1

Objetivo

Diseño de un lenguaje de programación para robots con bajas capacidades de cómputo, con fin educativo.

Diseño e implementación de una máquina virtual para dicho lenguaje, que debe ser implementado en alguna de las arquitecturas presentadas.

El lenguaje debe intentar estandarizar los diferentes kits de robótica.

1.1. Requerimientos

El lenguaje debe permitir expresar declarativamente o imperativamente el comportamiento de un robot. Se deben proveer abstracciones para el hardware periférico, de tal manera que se puedan definir nuevas abstracciones para incluir nuevo hardware. Se debe contemplar la posibilidad de compilar programas expresados en haskell mediante el paradigma de programación funcional reactiva.

A continuación se relevan las arquitecturas consideradas para el proyecto. Se relevan sus especificaciones y los trabajos relacionados que hay realizados en torno a la temática de éste proyecto.

Periféricos estándar:

1. Motor
2. Sensor de distancia
3. Comunicación (serial, inalambrica, usb, etc)
4. Acelerometro
5. Sensor de inclinacion
6. Boton, Interruptor
7. Sensor de sonido
8. Sensor de inclinacion

9. Gps
10. Sensor infrarrojo
11. Sensor de luz
12. Sensor de voltaje, etc
13. Brujula / compass
14. Parlante
15. Leds
16. Pantallas

1.2. Relevamiento de kits, placas, procesadores y arquitecturas

Prueba de citar [1]

1.3. TDL: A Task Description Language for Robot Control

Los sistema robóticos deben cumplir con objetivos al mismo tiempo que reaccionan ante el entorno y nuevas oportunidades. Generalmente, un robot debe coordinar actividades concurrentes, monitorear el ambiente y manejar excepciones. El lenguaje TDL es una extension de C++ que provee soporte sintactico para manejar tareas (Tasks). Las tareas se pueden descomponer, sincronizar, monitorear y manejar excepciones.

Un compilador traduce TDL a codigo puro C++ que utiliza una biblioteca independiente de las plataformas que maneja las tareas (Tasks).

El articulo introduce TDL, describe la representacion de arboles de tareas, (Task tree representation) y presenta algunos aspectos de su implementacion y su uso en la creación de robots moviles autónomos.

1.3.1. Introducción

Los sistemas robóticos, como robots móviles autónomos, necesitan alcanzar objetivos de alto nivel, y a su vez mantenerse reactivos a contingencias y nuevas oportunidades. Necesitan recuperarse satisfactoriamente de excepciones y administrar efectivamente sus recursos (Actuadores, sensores, poder de computo). Estas capacidades son conocidas como control a nivel de tarea (task-level control) y forman la base de la capa .ejecutiva"de la arquitectura en tres capas de control robotico.

Arquitectura en tres capas:

```
[Capa de planes]
<-- (status < , > planes) -->
[Capa ejecutiva]
<-- (datos sensados < , > comandos) -->
[Capa de comportamiento]
```

En estas arquitecturas, la capa de comportamiento interactúa con el mundo real, controlando actuadores y recolectando datos de los sensores. La capa de planes especifica en alto nivel, como conseguir objetivos y como manejar las interacciones. La capa ejecutiva expande objetivos abstractos a comandos de bajo nivel, ejecuta los comandos, monitorea su ejecución y maneja sus excepciones. A su vez reporta el estado a la capa de planes, el cual puede servir para tomar nuevas decisiones.

Desafortunadamente, la capa ejecutiva es compleja de desarrollar y de inspeccionar en busca de errores. (debug) El problema se da porque el control generalmente requiere que el robot realice tareas concurrentemente, tales como moverse y sensar, planear y ejecutar, manipular y monitorear, etc. Estas actividades concurrentes deben ser planificadas (scheduled) y sincronizadas, para evitar interacciones entre ellas o para coordinar actividades. Otra dificultad es que manejar una excepción muchas veces requiere control del flujo global. Por ejemplo, un robot encuentra un obstáculo, puede intentar esperar e intentar nuevamente, y si eso falla puede tener que volver a planear su camino, cambiar de objetivo, etc.

Usando lenguajes de programación convencionales para implementar tal control resultaría en código altamente NO linear que es muy dificil de entender, inspeccionar en busca de errores y mantener.

Pensando en esto, se diseñó el lenguaje de descripción de tareas TDL. TDL soporta descomposición de tareas, sincronización de subtareas, monitoreo de ejecución y manejo de excepciones. Un compilador transforma el codigo TDL en código C++ eficiente e independiente de la plataforma que invoca a la biblioteca TCM (Task control management) para controlar las tareas.

Se describen los árboles de tareas, una construcción semántica subyacente a TDL y TCM. Los árboles de tareas codifican la descomposición de tareas en subtareas, así como las restricciones de sincronización entre las mismas. Luego se describe el lenguaje en si mismo, y se muestra un ejemplo de uso en un robot autónomo de envío de correspondencia. Finalmente se muestra un poco de la implementación de TDL y TCM, así como herramientas desarroladas para soportar diseño y debugging.

1.3.2. Trabajo relacionado

TDL y TCM estan fuertemente influenciados en trabajos anteriores sobre Arquitectura de control de tareas. (TCA) TCA combina control de tareas (task-level control) y comunicacion entre procesos, usando pasaje de mensajes entre multiples procesos para obtener concurrencia. Se utiliza de TCA el concepto de arbol de tareas (task-tree), monitores de ejecucion y estructura jerarquica de manejo de excepciones. TDL y TCM extienden las estructuras de control de TCA para incluir capacidad de sincronizacion adicional, tal como ño comenzar una tarea hasta que pase cierto tiempo', 'terminar una tarea cuando otra se completa'y 'terminar una tarea luego de cierto periodo de tiempo'.

Asi como TCA, la biblioteca TCM es una lista de funciones que pueden ser invocadas para construir y coordinar arboles de tareas. TDL por otro lado, es un lenguaje completo, con su propia sintaxis (extension de C++). Otros investigadores han desarrollador lenguajes de control basados en control de tareas, para robots moviles y otros sistemas autonomos. Como TDL, la mayoria incluyen soporte para descomposicion de tareas, sincronizacion, monitoreo y manejo de excepciones.

1. RAP
2. PRS
3. ESL
4. Colbert

1.3.3. Árboles de tareas

Los arboles de tareas son la representacion basica subyacente a TDL y TCM. Un arbol codifica las relaciones padre/hijo y las restricciones de sincronizacion entre los nodos, y asocia manejadores de excepciones con nodos del arbol.

Los programas basados en TDL operan creando y ejecutando arboles de tareas. Casa nodo tiene asociada una accion, la cual es una pieza de codigo parametrizada. Una accion puede realizar calculos, agregar nodos al arbol de tareas, o realizar una accion en la realidad. Por ejemplo: 'moverse hacia adelante N metros', óbtener una imagen'. Ademas cada accion puede tener dos resultados, exito o error. (*succeed* or *fail*)

Como los árboles son generados dinámicamente, las acciones asociadas a los nodos pueden usar datos relevados actualmente para tomar decisiones acerca de que nodos agregar al arbol y como parametrizar sus acciones. Las acciones pueden incluir codigo iterativo, recursivo o condicional. El arbol resultante, sin embargo, siempre es un arbol simple: Cada arbol representa una traza de ejecucion unica del programa de control.

El mismo programa de control de tareas puede generar variedad de arboles de tareas de corrida en corrida.

Para aumentar los tipos de restricciones que pueden ser expresadas, distinguimos dos tipos de nodos: 'goals'(objetivos) y 'commands'(comandos). Los nodos de tipo *commands* tienen comportamiento ejecutable, son tipicamente las hojas del arbol. Los de tipo *goals* por otro lado, son utilizados para expandir el arbol de tareas, y representar tareas de mas alto nivel. Ej: ïr a posicion X", çentrarse en la puerta".

La accion asociada a un nodo *goal* es tipicamente un calculo que añade hijos al nodo. Mientras que los nodos *goal* pueden hacer calculos y agregar nodos, los nodos *commands* no pueden tener nodos *goal* como hijos.

El estado de un nodo puede ser *disabled*, *enabled*, *active* o *completed*. El nodo esta deshabilitado (*disabled*) si alguna restriccion de sincronizacion no ha sido satisfecha aun. Cuando todas esas restricciones son satisfechas, el nodo pasa a estar habilitado (*enabled*). El nodo pasa a activo (*active*) cuando la accion del nodo es invocada, un nodo puede estar habilitado pero no activo mientras no hay suficientes recursos computacionales o fisicos para correr la accion. Finalmente cuando la accion se ejecuta, puede terminar exitosamente o fallar, y pasa a estado completado. (*completed*)

Mientras que los nodos tienen un estado individual, usualmente es util referirse al estado de todo su subarbol. La expansion de un nodo, es el estado agregado de todos los nodos *goal* en el subarbol, incluyendo al nodo mismo. La ejecucion de un nodo, se refiere a todos los nodos *command* en su subarbol.

Como con un nodo, el estado de la expansion del nodo y de la ejecucion del nodo, puede ser exactamente uno de *disabled*, *enabled*, *active* o *completed*. Las transiciones ocurren en ese mismo orden.

Los estados corresponden a la nocion intuitiva de que un subarbol es expandido al manejar todos los objetivos del subarbol (goal) y que un nodo es ejecutado cuando todos sus comandos *command* fueron manejados.

Por ejemplo si la expansión de un nodo es completada esto implica que el manejo de todos los nodos objetivos en el arbol con raiz ese nodo estan completados.

Similarmente, si la ejecucion de un nodo esta deshabilitada, implica que el manejo de todos los comandos de ese arbol estan deshabilitados.

$(\forall N : Node)(Tree(N)/equiv N \cup_{c \in children(N)} Tree(c)$

TODO: Agregar las otras restricciones.

Mientras que un arbol de tareas impone restricciones entre nodos y sus hijos, por defecto no existen restricciones entre hijos. Por defecto se manejan concurrentemente. Sin embargo, a menudo se necesitan restricciones adicionales de sincronización para coordinar el comportamiento del robot de forma apropiada. Por ejemplo, la tarea de llevar el correo a un lugar incluye

1. Ir hasta el lugar
2. Centrarse en la puerta
3. Anunciar la presencia
4. Esperar que el correo sea recibido.

Las primeras 3 deben ser secuenciales, mientras que las ultimas dos pueden ser concurrentes.

TCM y TDL proveen restricciones de activacion y terminacion. Las de activacion indican que algun aspecto del nodo (handling, expansion, execution) no puede ser activado hasta que ocurra otro evento. El evento puede ser pasaje de tiempo, la transicion de estado de otro nodo, o un evento externo (un boton, etc). Por ejemplo uno podria hacer que el nodo A no pueda estar habilitado hasta que la ejecucion del nodo B se complete. (secuencial)

Similarmente, se puede decir que cierto nodo está deshabilitado hasta las 13:00 hs.

Las restricciones de terminación son similares. Indican que un nodo y todos sus hijos deben finalizar cuando cierto evento ocurra, si toma demasiado tiempo en completarse o alguna otra tarea comenzó su ejecución.

Permitiendo activación y terminación ser especificada en la -expansión-, -ejecución- o -estado- del nodo, se puede tener un control muy fino de cuando se activa cada parte del árbol. Por ejemplo si dos tareas deben ser secuenciales, pero la expansión de la segunda es computacionalmente cara. Se podría restringir la ejecución de la segunda tarea para que sea secuencial, pero permitir que su expansión sea concurrente, de manera que esté lista para ejecutarse cuando la primer tarea se complete.

Similarmente, se podria querer expandir una tarea antes que otra, pero ejecutarlas en orden opuesto. Por ejemplo, para hacer un viaje en avion, primero se determina que vuelo tomar, antes de decidir como llegar al aeropuerto, pero claramente la ejecucion es en otro orden. Podria ser

necesario crear restricciones entre diferentes niveles del arbol. Todas estas deciciones son expresibles facilmente con las restricciones de sincronizacion de arboles de tareas.

Un *monitor* es un tipo de nodo cuya accion puede ser invocada repetidamente. Luego que un monitor es habilitado, eventos pueden activarlo, cada uno causa una invocacion separada de su accion.

Luego que se activa un numero determinado de veces, el monitor pasa a estado *completado*. Los eventos que pueden activar un monitor incluyen el pasaje del tiempo, una transicion en otro nodo o un evento externo. Una accion de un monitor puede disparar un evento, que significa que alguna condicion fue detectada, y puede ser usada para determinar cuando completar el monitor. Por ejemplo, si quisieramos hacer que un robot se mueva hasta que vea una marca especifica. Se podria hacer que el monitor se active cada 200 milisegundos, corriendo concurrentemente con "navegar por el corredor", y la tarea navegar con una restriccion de terminar cuando se complete el monitor. Cuando el monitor ve la marca dispara un evento, y causa que la tarea "navegar por el corredor"termine.

Los manejadores de .^{Ex}cepciones.^{es}tan asociados con un nodo dado en el arbol y por un motivo (string definida). Cuando una accion falla (ej: un motor se sobrecalienta o no se obtiene un camino posible en un plan), especifica un motivo. TCM conduce una busqueda hacia arriba en la jerarquia buscando el primer manejo de excepcion que tenga el mismo motivo. El manejador de excepcion es invocado, y puede intentar recuperarse del problema agregando nuevos nodos o finalizando los nodos existentes. Alternativamente puede decir que no es capaz de manejar la excepcion, y la excepcion sigue subiendo en la jerarquia. El mecanismo es similar al çatch and throw"de C++, etc. El arbol permanece intacto durante este mecanismo, el manejador de excepciones debe decidir que partes del arbol modificar para recuperarse.

1.3.4. TDL

TDL es una extension de C++ que facilita la creacion, sincronizacion y manipulacion de arboles de tareas. Las tareas se definen similarmente a las funciones de C++. El nombre de la tarea es precedido por un identificador de Clase (Goal, Command, Monitor, Exception) y seguido por sus argumentos, restricciones opcionales, y el cuerpo de la tarea. Las tareas, a diferencia de las funciones C++, no tienen valor de retorno.

El cuerpo puede incluir codigo C++ arbitrario, con ciertas restricciones. Primero, las tareas deben ser globales, no pueden ser definidas dentro de clases. Similarmente, las funciones y metodos de clase, no pueden ser definidos dentro de una tarea, aunque el mismo archivo TDL puede contener ambas tarea y funciones. Finalmente, el uso de "gotoτ formas similares de transferencia del control, no son permitidos.

El comando "spawn.^{es} utilizado para aniadir un nodo hijo al arbol. "spawn"no es bloqueante, una subtarea hija podria no ser manejada antes que el control sea regresado al padre. Las tareas creadas con el comando, pueden sincronizarse con la clausula "with". "with"hace que "spawn"sea bloqueante, el control no se regresa al padre, hasta que la tarea hija sea manejada junto con todos sus descendientes.

TDL define restricciones comunes. Por ejemplo "secuential execution ¡node¿.^{eq}uivale a "disable execution until ¡node¿execution completed". .^{ex}pand first"no permite ejecucion hasta que la expansion se complete, y "delay expansion"no permite expansion hasta que ejecucion no se permita.

Las palabras claves "self", "parent", "previous"pueden aparecer para referirse a la tarea misma, a la tarea padre o previa. La tarea previa se decide en tiempo de ejecucion, ya que es dificil saber cual seria la tarea previa antes, esto se determina con codigo agregado por el compilador TDL. Las tareas se pueden etiquetar si multiples tareas fueron creadas:

```
t1: spawn a(1)
t2: spawn a(2)
spawn b(3) with
  disable until t1 execution completed,
  disable expansion until t2 handling active;
```

La etiqueta "previous"puede ser util, pero puede llevar a codigo dificil de entender, en estos casos deberian usarse etiquetas.

Las restricciones pueden ser aplicadas a varias tareas. TDL lo soporta usando la sintaxis $with(< constraint >)do< body >$. Las restricciones son aplicadas a todas las tareas que se creen en ¡body¿.

Para sentencias with do anidadas, las restricciones son aplicadas al with do interno como una entidad, como si fuera una tarea separada.

En el cuerpo de una tarea puede contener sentencias *succeed* y *fail ¡motivo¿*. Ambas hacen que la tarea se marque como completada, y *fail* lanza una excepcion. Los manejadores de excepciones se definen analogamente a los objetivos y comandos (*goal* y *command*). Los manejadores pueden contener la sentencia "bypass", que indican que otro manejador deberia ser encontrado para manejar la excepcion. Los manejadores de excepciones son asociados con los nodos de tareas usando la sentencia $withexception(< motivo >:< manejador > ...)do< body >$

Por ejemplo:

```
Goal navigateToLocation(double x, double y) {
  with exception
    ("Overheating": handleOverheating(x, y),
     "no path": handlePlannerFailure()) do {

     }
}
```

Los monitores son definidos y lanzados de la misma forma que los objetivos y los comandos. Sin embargo, para los monitores se puede especificar restricciones globales. Por ejemplo "max triggers = ¡num¿", "max activations = ¡num¿τ "period = ¡time¿".

1.3.5. Implementación

TDL esta implementado usando un compilador que transforma las definiciones de tareas en C++ puro que incluye invocaciones a la biblioteca TCM. Este codigo puede ser compilado usando un compilador estandar de C++ y vinculado con la biblioteca TCM. Esto tiene variedad de ventajas. Primero se puede tomar ventaja del compilador de C++ que produce codigo independiente de la plataforma, optimizado, y tener control de tareas eficiente. Luego esto permite que el codigo TDL reutilice codigo C y C++ existente, incluyendo funciones que usan TCM directamente.

TDLC es el programa que transforma el codigo TDL, esta escrito en Java, con el parser escrito en JavaCC. El archivo TDL es parseado y se crea una red de objetos Java que tienen una correspondencia 1-1 con las definiciones de tareas, las sentencias TDL, codigo C++ y el archivo mismo. Cada objeto es capaz de imprimirse a si mismo, en el formato TDL original o codigo C++ traducido.

La traduccion a C++ es trivial para las partes de la tarae que ya son sentencias C++. La creacion de tareas (task spawn) y las restricciones de sincronizacion son traducidad directamente a codigo C++ que invoca las funciones correspondientes en TCM. Cada definicion de tarea TDL se transforma en una Clase C++. La clase incluye variables para cada argumento formal de la tarea y un metodo para invocar el cuerpo (¡body¿) de la tarea. Se transforma en la accion del nodo del arbol. Ademas archivos de definicion (header) y funciones son generadas para crear un nuevo nodo de ese tipo. Creando la accion del nodo e invocando la tarea como si fuese una funcion C estandar.

Muchas clases especializadas se usan para ayudar a manejar los nodos de tareas y las restricciones de sincronizacion. En particular el objeto $_{T}DL_{H}andleManager$ se usa para mapear los nombres de tareas con las referencias al nodo correspondiente en el arbol. Este mismo objeto se usa para mantener referencia al anidado de "with/doτ cuando los nodos se crean, para poder determinar que nodo se considera previo de otro.

La biblioteca TCM esta implementada en C++. Una jerarquia de clases se define para los tipos de nodos de arbol. Cada nodo especifica su padre, hijo, accion asociada, excepciones asociadas, estado actual propio, de su ejecucion y de su expansion, asi como listas de restricciones de sincronizacion de las que depende, y de las que otro depende en relacion a el nodo. Cuando un nodo del arbol cambia de estado, envia una senial a los nodos que estan esperando por su transicion. Un "manejador de agenda"(agenda manager) encola y despacha estas seniales, invoca las acciones de los nodos activos, y envia seniales a los nodos que esperan por un tiempo determinado. TCM puede ser usado en modo DEBUG y loguear todas las transiciones.

Al momento del paper, TCM no soporta Multitasking real. Lo hace mucho mas portable, pero limita la concurrencia real que pueda soportar.

1.3.6. Leer

- 15

  task-level control.

- 11

  soporte para manejo de recursos.

13,14,15 Task control architecture.

14 estructura jerarquica de manejo de excepciones.

13 restricciones a diferentes niveles del arbol.

15	manejo de Excepciones
12	ControlShell, provee control de tiempo real.
8, 6	RAP, PRS

Bibliografía

[1] Pacheco, G., and Turner, W. Titulo de prueba. *The journal of journals* (2014), 11–15.