A Task Description Language for Robot Control

Reid Simmons and David Apfelbaum

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Robot systems must achieve high level goals while remaining reactive to contingencies and new opportunities. This typically requires robot systems to coordinate concurrent activities, monitor the environment, and deal with exceptions. We have developed a new language to support such task-level control. The language, TDL, is an extension of C++ that provides syntactic support for task decomposition, synchronization, execution monitoring, and exception handling. A compiler transforms TDL into pure C++ code that utilizes a platform-independent task management library. This paper introduces TDL, describes the task tree *representation that underlies the language, and presents some aspects of its implementation and use in an autonomous mobile robot.*

Introduction

Robot systems, such as autonomous mobile robots, need to achieve high level goals while remaining reactive to contingencies and new opportunities. They need to recover gracefully from exceptions and effectively manage their resources (such as actuators, sensors, and computation). These capabilities are referred to as *task-level control* [15], and they form the basis of the *executive* layer of modern three-tiered robot control architectures [1, 3, 4, 10]. In such architectures (Figure 1), the *behavior* (real-time control) layer interacts with the physical world, controlling actuators and collecting sensor data. The *planning* layer specifies, at an abstract level, how to achieve goals and how to deal with goal interactions. The *executive* layer mediates between the symbolic level of the planner and the continuous level of the behaviors. It expands abstract goals into low-level commands, executes the commands, monitors their execution, and handles exceptions.

Unfortunately, task-level control programs are often difficult to develop and debug. One problem is that effective task-level control often requires that the robot do things concurrently, such as moving and sensing, planning and executing, manipulating and monitoring, etc. These concurrent activities often need to be scheduled and synchronized, either to avoid interactions or to coordinate activities. Another difficulty is that exception handling often involves non-local flow of control. For example, if a robot encounters an unexpected obstacle, it might first try the move again (the obstacle may have moved). If that fails, it might replan its path, switch to another goal, etc.

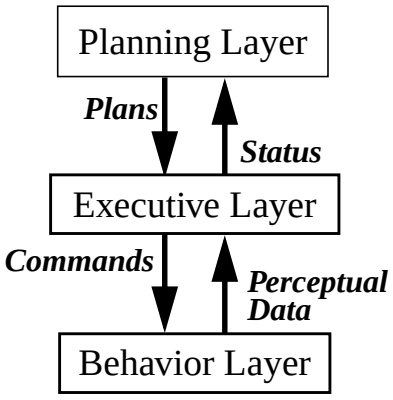

Figure 1: Three-Tiered Control Architecture

Using conventional programming languages to implement such task-level control functions would result in highly non-linear code that is often difficult to understand, debug, and maintain. To address this, we have designed TDL (Task Definition Language), an extension of C++ that simplifies the development of robot control programs by including explicit syntactic support for task-level control capabilities. TDL directly supports task decomposition, fine-grained synchronization of subtasks, execution monitoring, and exception handling (support for resource management [11] is planned). We have developed a compiler that transforms TDL code into efficient, platform-independent C++ code that invokes a Task Control Management (TCM) library to manage task-control aspects of the robot system.

The following section presents related research in languages for task-level control. We then describe *task trees*, the semantic construct underlying TDL and TCM. Task trees encode the hierarchical decomposition of tasks into subtasks, as well as synchronization constraints between tasks. We then describe the language itself, and illustrate it with a simplified example of its use in an autonomous delivery robot [16]. Finally, we present overviews of the TDL and TCM implementations, as well as tools that we are developing to further support the design and debugging of task-level control programs.

Related Work

TDL and the TCM library are both heavily influenced by our earlier work on the Task Control Architecture (TCA) [13, 14, 15]. TCA combines task-level control and inter-process communication, using message passing between multiple processes to achieve concurrency. Aspects of TCA that are maintained include the underlying concept of a *task tree* (see next section), execution monitors, and the approach to hierarchical structuring of exception handlers [14]. TDL and TCM extend the TCA control constructs to include additional synchronization capabilities, such as not starting a task until a given time, terminating a task when another task completes, and terminating a task after a certain period of time.

Like TCA, the TCM library is a collection of function calls that can be invoked to build and coordinate task trees. TDL, on the other hand, is a full-fledged language, with its own syntax (an extension of C++). Other researchers have also developed task-level control languages for mobile robots and other autonomous systems. Like TDL, most include explicit support for task decomposition, synchronization, monitoring and exception handling. RAP (Reactive Action Packages) was designed to support reactive planning and execution and also to be used as the representation language for a general-purpose planner [3]. RAP uses a Lisp-based interpreter to manage a *task net* and to interface with behavioral *skills*. RPL (Reactive Plan Language), which was influenced by RAP, incorporates a richer set of control constructs [8]. Like RAP, RPL was designed to be used by a planner, and runs using a Lisp-based interpreter.

PRS (Procedural Reasoning System) is based around the concept of a procedural reasoning expert [6]. PRS facilitates deciding what actions an agent should be doing at any given time. Both Lisp-based and C-based interpreters for PRS have been implemented. PRS, like RAP, is tightly integrated with a "world model" knowledge base that is used to identify opportunities, exceptions, and when to transition between tasks. TDL does not make this ontological commitment: A separate knowledge base could be integrated, but is not mandated. We feel that this gives developers more flexibility in deciding how to design their systems, without precluding such architectural decisions.

ESL (Executive Support Language) is the language most closely related to TDL, and TDL is influenced by many of the ESL design concepts [5]. Like TDL, ESL is an extension of an existing language (in this case, Lisp). ESL is implemented as a set of macros that expand into Common Lisp and invoke Lisp's multi-tasking library. In addition to the usual task-level control constructs, ESL provides for resource management and supports a Prolog-based data base. ESL is currently being used in the NASA New Millennium Remote Agent [10].

Colbert has a C-like syntax, but does not support the full C language [7]. Colbert supports concurrency and iteration, but does not provide explicit support for exception handling. It has well-defined semantics based on finite state automata, which makes it easy to determine how the system will behave. It has a compiler, but is mainly intended for interactive use, via an interpreter.

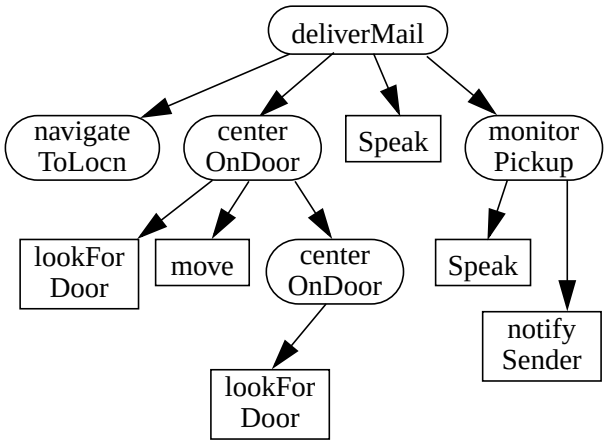

Figure 2: Example Task Tree

One major difference between TDL and other task-control languages is that TDL includes a wider range of task synchronization constructs. For instance, in TDL one can state that the system should completely expand a task tree down to its leaf nodes (executable commands) before beginning to execute any of those commands. While other languages, notably ESL, can also express such concepts, the range of control constructs in TDL makes it possible to state intricate control strategies fairly straightforwardly.

Task Trees

Task trees are the basic representation underlying TDL and TCM. A task tree (Figure 2) encodes parent/child relationships and synchronization constraints between nodes, and associates exception handlers with nodes in the tree. TDL-based control programs operate by creating and executing task trees. Each task tree node has an *action* associated with it, which is essentially a parameterized piece of code. An action can perform computations, dynamically add child nodes to the task tree, or perform some physical action in the world, such as "move forward N meters" or "acquire an image". In addition, actions can either *succeed* or *fail*.

Note that since task trees are generated dynamically, the actions associated with nodes can use current perception data to make decisions about what nodes to add to the tree and how to parameterize their actions. Actions can include conditional, iterative, and even recursive code. The resulting task tree, however, is always a simple tree: Each task tree represents a single execution trace of the control program. Thus, the same task-control program can generate widely different task trees from run to run.

To increase the types of synchronization constraints that can be expressed, we distinguish two types of nodes: *goals* and *commands* (two other types of nodes, *monitors* and *exceptions*, are described later in this section). Command nodes are intended to be executable behaviors, and are typically the leaves of a task tree. Goal nodes, on the other hand, are used to expand the task tree, and represent higher-level tasks, such as "navigate to location X" or "center on a doorway". The action associated with a goal node is typically a computation that adds children to that node. While goal nodes can have both goals and commands as children, commands cannot have goal nodes as children.

The state of an individual task tree node (referred to as its *handling*) can be either *disabled*, *enabled*, *active* or *completed*. The *handling* of a node is *disabled* if there are synchronization constraints (see below) that have not yet been satisfied. When all such constraints are satisfied, the *handling* transitions to *enabled*. The *handling* then transitions to *active* and the node's action is invoked (a node can be *enabled* but not *active* if there are insufficient computational or physical resources to run the action). Finally, when the action succeeds or fails, the *handling* transitions to *completed*.

While the *handling* refers just to the state of an individual node, it is often useful to refer to the state of its whole subtree. The *expansion* of a node refers to the aggregate state of all goal nodes in the subtree, including the node itself ("deliverMail", "navigateToLocn", "centerOnDoor", and "monitorPickup" in Figure 2). The *execution* of a node refers to all command nodes in the subtree ("lookForDoor", "speak",, "move" and "notifySender" in Figure 2). As with a node's *handling*, the state of a node's *expansion* and *execution* can be exactly one of *disabled*, *enabled*, *active*, or *completed* (and state transitions occur in that order).

The various states correspond to the intuitive notion that a subtree is expanded by handling all the goal nodes in the tree, and that a node is executed when all its commands have been handled. For instance, if a node's *expansion* is *completed*, that implies that the *handling* of all goal nodes in the tree rooted at that node are *completed*. Similarly, if a node's *execution* is *disabled*, that implies that the *handling* of all commands in the tree are *disabled*. Figure 3 presents formal relationships between the *expansion* and *execution* of a node and the *handling* of the nodes in its subtree.[1]

While the task tree imposes constraints between parent and children nodes, by default there are no constraints between children nodes themselves (i.e., by default they are all handled concurrently). Often, however, additional synchronization constraints are needed to coordinate robot behaviors in appropriate ways. For example, the task of

1. For completeness, the *expansion* of a command node is defined to always be in the *completed* state.

$$\forall(N{:}node)(Tree(N) \equiv \{N\} \cup \bigcup_{c \in children(N)} Tree(c))$$

$$\forall(N{:}goal)Expansion(N, Disabled) \Rightarrow [\forall(n \in Tree(N))goal(n) \Rightarrow Handling(n, Disabled)]$$

$$\forall(N{:}goal)\exists(n \in Tree(N))goal(n) \wedge Handling(n, Enabled) \Rightarrow (Expansion(N, Enabled) \vee Expansion(N, Active))$$

$$\forall(N{:}goal)Expansion(N, Active) \Rightarrow [\exists(n \in Tree(N))goal(n) \wedge (Handling(n, Active) \vee Handling(n, Completed))] \wedge \neg Expansion(N, Completed)$$

$$\forall(N{:}goal)Expansion(N, Completed) \Rightarrow [\forall(n \in Tree(N))goal(n) \Rightarrow Handling(n, Completed)]$$

$$\forall(N{:}node)Execution(N, Disabled) \Rightarrow [\forall(n \in Tree(N))command(n) \Rightarrow Handling(n, Disabled)]$$

$$\forall(N{:}node)[\exists(n \in Tree(N))command(n) \wedge Handling(n, Enabled)] \Rightarrow (Expansion(N, Enabled) \vee Expansion(N, Active))$$

$$\forall(N{:}node)Execution(N, Active) \Rightarrow [\exists(n \in Tree(N))command(n) \wedge (Handling(n, Active) \vee Handling(n, Completed))] \wedge \neg Execution(N, Completed)$$

$$\forall(N{:}node)Execution(N, Completed) \Rightarrow [\forall(n \in Tree(N))Handling(n, Completed)]$$

Figure 3: Expansion and Execution Semantics

delivering mail to some location may involve 1) navigating to that location, 2) centering on the doorway, 3) announcing one's presence, and 4) waiting for the mail to be taken. The first three of these should be done sequentially, while the last two can be done concurrently.

TCM and TDL provide both *enablement* and *termination* synchronization constraints. Enablement constraints indicate that some aspect of a node (its *handling*, *expansion* or *execution*) cannot become *enabled* until some other event occurs. The event can be either the passage of time, a specific state transition of another node, or some external event (e.g., triggered by a button push in a GUI). For instance, one can state that the *execution* of node **A** cannot become enabled until the *execution* of node **B** is completed (i.e., sequential execution). Similarly, one can state that the *handling* of some node is disabled until 1PM. Termination constraints are similar. Here, the constraints indicate that a node and all its children are to be terminated when some event occurs, such as if the task takes too long to complete, or some other task has begun execution.

Allowing for enablement and termination constraints to be specified in relationship to each of the *handling*, *expansion* and *execution* of nodes facilitates very fine-grained task-level control. For instance, suppose two tasks must occur sequentially, but expanding the second task is computationally expensive. Then, one might want to

constrain the *execution* of the second task to be sequential, but allow it to be expanded concurrently, so that it is ready for execution when the first task completes. Similarly, one might want to expand one task before another, but execute them in the opposite order. For instance, in making plans for air travel, one typically determines what flight to take before deciding how to get to the airport, but clearly the execution is in the opposite order. One might even need to place constraints between nodes on different levels of the tree [13]. All these types of control decisions are easily expressible using the task tree synchronization constraints.

A *monitor* is a type of task tree node whose action can be invoked repeatedly. After a monitor is *enabled*, events can *activate* it, which cause a separate invocation of its action. After a (user-specified) number of activations, the monitor's *handling* becomes *completed*. Events that can activate a monitor include the passage of time, a state transition of some other node, or an external event. A monitor's action can issue a *trigger* event (which roughly corresponds to some condition being observed), which can be used to determine when to complete the monitor. For instance, suppose we want the robot to travel down a hallway until it sees a specified landmark. We could have monitor that is activated every 200 milliseconds running concurrently with a "navigate down corridor" task that is constrained to terminate upon completion of the monitor. When the monitor sees the landmark it issues a trigger event, which causes it to complete and the "navigate" task to terminate.

Exceptions are treated as described in [15]. Exception handlers are associated with a given node in the task tree and a specific *reason* (some user-defined string). When an action *fails* (e.g., because a motor overheats, or a planner cannot find a valid path), it specifies the reason for the failure. TCM then conducts a search up the tree for the first exception handler that matches the given reason. The exception handler is then invoked, and it can try to recover from the problem by adding new nodes or terminating existing nodes. Alternately, it can issue a *bypass*, which indicates that it is unable to handle the exception. In this case, the search for a matching exception handler continues up the tree. This hierarchical structuring of exceptions is similar to "catch and throw" mechanisms used in languages such as C++, Lisp and Ada. A key difference is that here the control stack is not popped when an exception handler is invoked. The task tree remains intact, and it is up to the exception handler to decide what parts of the task tree to modify in order to recover from the exception.

TDL

TDL is an extension of C++ that facilitates the creation, synchronization, and manipulation of task trees. Tasks are defined in a manner similar to C++ functions: The name of a task is preceded by a class identifier (Goal, Command, Monitor, Exception) and followed by its arguments, optional top-level constraints, and the task body (Figure 4). Unlike C++ functions, tasks do not have a return value.

```
Goal deliverMail (int room)
{
  double x, y;
  getRoomCoordinates(room, &x, &y);
  spawn navigateToLocn(x, y);
  spawn centerOnDoor(x, y)
      with sequential execution previous,
        terminate in 0:0:30.0;
  spawn speak("Xavier here with your mail")
      with sequential execution centerOnDoor,
        terminate at monitorPickup completed;
  spawn monitorPickup()
      with sequential execution centerOnDoor;
}

Goal centerOnDoor (double x, double y)
      delay expansion
{
  int whichSide;
  spawn lookForDoor(&whichSide) with wait;
  if (whichSide != 0) {
      if (whichSide < 0)
        spawn move(-10); // move left
      else
        spawn move(10); // move right
      spawn centerOnDoor(x, y)
        with disable execution until
          previous execution completed;
  }
}
```

Figure 4: Sample Task Definitions (Simplified)

The task body can contain arbitrary C++ code, with certain restrictions. First, tasks must be globally scoped (they cannot be defined inside C++ classes). Similarly, functions and class methods cannot be defined inside a task body, although the same TDL file may contain both task and function definitions. Finally, non-local, non-continuous transfer of control is not permitted, prohibiting the use of "`goto`" and similar functionality such as "`longjmp`".

The "`spawn`" statement (Figure 4) is used to add a child node to the task tree. "`spawn`" is non-blocking, in that the child subtask may not actually be handled by the time control returns to the parent task. Spawned tasks can be synchronized using the "`with`" clause. Figure 5 presents the syntax for the currently defined set of constraints. Most correspond directly to the synchronization constraints described in the previous section, and their meanings should be fairly obvious. The "`wait`" constraint makes "`spawn`" blocking, so that control is not returned until the spawned task, and all of its descendants, have been handled.

<simple-constraint>:
expand first | **delay expansion**

<constraint>:
<constraint> , <constraint> | <simple-constraint>
| **sequential** <constraint-option> [<tag>]
| **serial** [<tag>] | **parallel** | **wait**
| **disable** [<constraint-option>] **until** <event>
| **disable** [<constraint-option>] **until** <absolute-time>
| **disable** [<constraint-option>]
for <relative-time> [**after** <event>]
| **terminate at** <event>
| **terminate at** <absolute-time>
| **terminate in** <relative-time> [**after** <event>]

<event>:
<tag> [<constraint-option>] <state>

<tag>:
<task name> | **self** | **parent** | **previous**

<constraint-option>:
handling | **expansion** | **execution**

<state>:
enabled | **active** | **completed**

Figure 5: TDL Synchronization Constraints

TDL defines some shorthands for commonly used constraints. For instance, "`sequential execution <node>`" is equivalent to "`disable execution until <node> execution completed`". Since the "`expand first`" (do not enable execution until expansion is completed) and "`delay expansion`" (do not enable expansion until execution is enabled) shorthands refer only to the node being spawned, they can also appear as top-level task constraints. This is illustrated in the definition of "`centerOnDoor`" in Figure 4. Note that combining "`delay expansion`" and "`sequential execution`" constraints produces totally serial behavior.

The `<tag>` part of a constraint (Figure 5) can be the name of a spawned task that appears within the task body, or the keywords "`self`", "`parent`", or "`previous`". If multiple tasks of the same name are spawned, the referential ambiguity is resolved by using explicit labels:[2]

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

The tag "`self`" refers to the task that is being spawned and the tag "`parent`" refers to the enclosing task. The tag "`previous`" is more complex to interpret. Spawned tasks may be embedded within iterative or conditional code, making it impossible to determine the previous task statically, at compile time. Therefore, this determination is made dynamically, at run time, using code added by the TDL compiler. While being able to refer to the "`previous`" spawned task is often useful, in some cases it may lead to hard-to-understand, unpredictable code. In such situations, one should use explicitly named tags.

Sometimes, the same synchronization constraint must be applied to a number of tasks. TDL supports this through the "`with (<constraint>) do {<body>}`" syntax. The constraints of a "`with/do`" statement are applied to each spawned task in the `<body>`. For nested "`with/do`" statements, the constraints of the outer "`with/do`" are applied to the inner "`with/do`" as a single entity, as if the inner "`with/do`" were itself a separate task. For instance, the "`deliverMail`" task in Figure 4 could be written as:

```
getRoomCoordinates(room, &x, &y);
with (sequential execution) do {
  spawn navigateToLocn(x, y);
  spawn centerOnDoor(x, y)
    with terminate in 0:0:30.0;
  with (parallel) do {
    spawn speak("Xavier here with your mail")
      with terminate at monitorPickup completed;
    spawn monitorPickup();
  }
}
```

which indicates that the "`speak`" and "`monitorPickup`" tasks can execute concurrently, and that both execute sequentially after task "`centerOnDoor`", which in turn executes sequentially after "`navigateToLocn`".

A task body can also contain the statements "`succeed`" and "`fail <reason>`". Both cause the task to complete handling, with the "`fail`" statement raising an exception. Exception handlers are defined analogously to goal and command tasks. Only exception handlers, however, can include the "`bypass`" statement, which indicates that another exception handler should be found to handle the current exception. Exception handlers are associated with task tree nodes using the "`with exception (<reason>: <handler> ...) do { <body> }`" statement. For instance:

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

provides the navigation task with handlers for both the "`Overheating`" and "`no path`" exceptions.

Monitors are defined and spawned in much the same way as goal and command tasks. Monitors, however, can specify additional top-level task constraints, including "`max triggers = <num>`", "`max activations = <num>`", and "`period = <time>`". Within the body

2. TDL also includes syntax for referring to a specific task, or set of tasks, that are spawned within an iterative (`for`, `while`, or `do`) statement.

```
Monitor monitorPickup ()
      max triggers = 1, max activations = 15,
      period = 0:0:1.5
{
  if (mailIsGone()) {
      trigger;
      with (parallel) do {
        spawn speak("Thank you");
        spawn notifySender();
      }
  }
}
```

Figure 6: Monitor for Delivery Task (Simplified)

of a monitor task, the statement "`trigger`" can be used to signal a trigger event (Figure 6).

Implementation

TDL is implemented using a compiler that transforms the task definitions into pure C++ code that include calls to the TCM library. This code can then be compiled using any standard C++ compiler and linked with the TCM library. This approach has several advantages. First, we can take advantage of widespread, highly optimized C++ compilers to produce platform-independent, efficient task-level control code. Second, this enables TDL code to easily interface with existing C and C++ code, including functions that use the TCM library directly.

TDLC, the program that transforms TDL code, is written in Java, with the parser written in JavaCC. As a TDL file is parsed, a network of Java data-objects are created that have a direct one-to-one correspondence with TDL task definitions, TDL statements, C++ code, and even the file itself. Each data-object is capable of printing itself out, either in the original TDL format or translated C++ code.

The translation to C++ is trivial for the parts of the task definition that are already C++ statements. TDL-specific statements, such as task definition headers, `spawn`s, and `with/do`s, are somewhat more complex. Task spawns and synchronization constraints are translated directly into C++ code that invokes the corresponding TCM functions. Each TDL task definition is translated into a C++ class. The class includes variables for each formal argument of the task and a method to invoke the body of the task (which becomes the task tree node's *action*). In addition, header files and functions are generated for allocating a new task tree node of that type, creating the node's action, and invoking the task as if it were a standard C function.

Several specialized classes are used to help manage task tree nodes and synchronization constraints. In particular, the `_TDL_HandleManager` object is used to map between task names (such as "`navigateToLocn`") and references to the corresponding task tree nodes. This same object is also used to keep track of `with/do` nesting and when task tree nodes are allocated, so as to determine which node(s) are considered to be "previous".

The TCM library is written in C++. A hierarchy of classes is defined for the various types of task tree nodes. Each node specifies its parent, children, associated action, associated exceptions, current state of its *handling*, *expansion* and *execution*, and lists of the synchronization constraints it depends on and those that are dependent upon it. When a task tree node changes state, it signals all nodes that are waiting for that transition, as specified by the "`disable`" and "`terminate`" constraints (Figure 5). An *agenda manager* queues and dispatches these signals, invokes the actions of enabled nodes, and signals nodes that are waiting on a given time. TCM can also be instructed to log all state transitions, for use in debugging.

We have also integrated the TCM library with ControlShell (produced by Real-Time Innovations), which provides for real-time control [12]. The two packages communicate via message passing, using RTI's NDDS system [9]. Together, these packages form the bottom two layers of a standard three-tiered control architecture (Figure 1).

Currently, the TCM implementation does not support true multi-tasking. While this makes the TCM library fairly portable, it also limits the amount of true concurrency the system can support. Once the basic concepts have been proven, we intend to create tailored implementations of the TCM library that take advantage of multi-tasking capabilities provided by various operating systems.

Tool Support and Future Work

We are developing several tools to support the design and debugging of task-level control programs. VDT (Visual Design Tool) is a graphical programming environment, written in Java, that supports the design of TDL code. VDT enables programmers to define TDL tasks using a mixed textual/graphical interface (Figure 7). VDT can also produce HTML documents from TDL code, to facilitate browsing for existing task definitions.

To facilitate debugging, we are developing visualization tools that use the log files produced by TCM. These tools will enable users to view the hierarchical structure of task trees, to see the changes in node state over time, and to identify constraints between nodes that are not yet satisfied. We are also using temporal logic to model state transitions and synchronization constraints between task tree nodes. The goal is to use model checking [2] to formally validate properties of task-level control programs, such as liveness and safety, and possibly to verify the implementation of the TCM library itself.

Other research includes integrating TDL and TCM with resource management capabilities [11], another important aspect of task-level control. We are also looking at transforming other existing task-level control languages

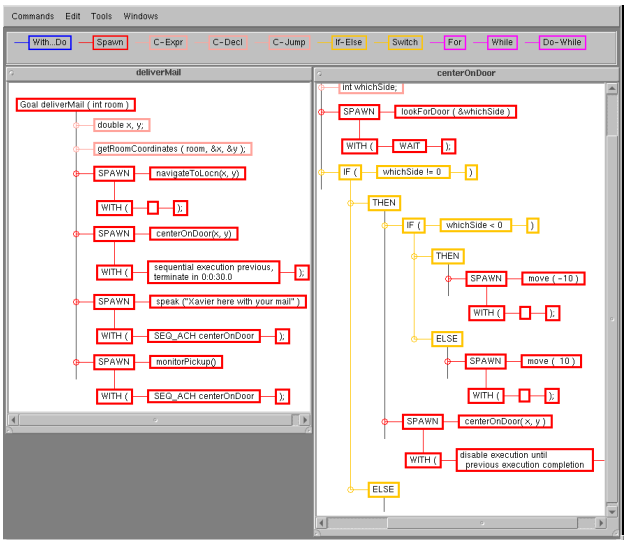

Figure 7: Visual Design Tool (VDT)

into TDL, both to demonstrate the expressive power of TDL and to provide compiled, C++-based implementations of those languages. Our initial exploration, which focuses on the RAP [3] language, shows promising early results.

Conclusions

This paper has presented TDL, a new language for specifying task-level robot control. TDL is an extension of C++ that includes explicit synchronization constructs to support task decomposition, synchronization, execution monitoring and exception handling. We have described in detail the concept of task trees, which underlies all of TDL, including how it supports fine-grained control over task synchronization. We introduced the syntax of TDL, illustrated it with a simplified mobile robot example, and briefly described how TDL code is transformed into pure C++ code that utilizes calls to our task management library.

Designing a new language is tricky. A good language should embody enough constraints so as to guide developers along the "correct" path, without mandating decisions that may be unwarranted. Simple robot behavior should be simple to state in the language, while complex behaviors should still be expressible, in some manner. It is difficult to make these trade-offs correctly *a priori*. Our extensive experience with TCA, coupled with our familiarity with other task-level control languages, especially ESL and RAP, has provided a good basis for justifying our choices. The real proof, however, is in the use. To this end, we are starting to use TDL in various applications, including the Xavier mobile robot [16]. Through this experience, we expect to refine and extend TDL, the TCM library, and the various support tools. Our aim is to make developing complex robot control programs as easy as conventional software. The payoff is cheaper, more reliable, and more effective robot systems.

Acknowledgments

G.R. Srinivasan and Greg Whelan have contributed to this effort. Erann Gat provided many insightful comments on an earlier draft. This research has been partially supported by NASA, under grant NAGW-5113.

References

[1] P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. A Proven Three-Tiered Architecture for Programming Autonomous Robots, *Journal of Experimental and Theoretical Artificial Intelligence*, **9:2**, 1997.

[2] E. Clarke, E. Emerson and A. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, *ACM Transactions on Programming Languages and Systems*, **8:2**, pp 244-263, 1986.

[3] R. James Firby. An Investigation into Reactive Planning in Complex Domains. Proc. National Conference on Artificial Intelligence, pp 202-206, Seattle, WA, July, 1987

[4] E. Gat. Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots, Proc. National Conference on Artificial Intelligence, pp 809-815, San Jose, CA, July 1992.

[5] E. Gat. ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents, Proc. AAAI Fall Symposium on Plan Execution, Boston MA, October 1996.

[6] M. Georgeoff and A. Lansky. Reactive Reasoning and Planning. Proc. National Conference on Artificial Intelligence, pp 972-978, Seattle, WA, July, 1987.

[7] K. Konolige. "COLBERT: A Language for Reactive Control in Saphira", Proc. German Conference on Artificial Intelligence, Freiburg, Germany, 1997.

[8] D. McDermott. Transformational Planning of Reactive Behavior, Tech Report YALEU/DCS/RR-941, Yale University, 1994.

[9] G. Pardo-Castellote, S. Schneider and M. Hamilton. Publish-Subscribe "Push" Technologies for Real-Time Applications, Proc. Real-Time Systems Symposium, December 1997.

[10] D. Bernard and B. Pell. Designed for Autonomy: Remote Agent for the New Millennium Program. Proc. i-SAIRAS '97, Tokyo Japan, October 1997.

[11] B. Pell, G. Dorais, C. Plaunt and R. Washington, "The Remote Agent Executive: Capabilities to Support Integrated Robotic Agents", Proc. AAAI Spring Symposium on Integrated Robotic Architectures, Palo Alto, CA, March 1998.

[12] S. Schneider, V. Chen, G. Pardo-Castellote. ControlShell: A Real-Time Software Framework, Proc. International Conference on Robotics and Automation, San Diego CA, May 1994.

[13] R. Simmons. Concurrent Planning and Execution for Autonomous Robots, *IEEE Control Systems*, **12:1**, pp 46-50, 1992.

[14] R. Simmons. Monitoring and Error Recovery for Autonomous Walking, Proc. IEEE International Workshop on Intelligent Robots and Systems, pp 1407-1412, July 1992.

[15] R. Simmons. Structured Control for Autonomous Robots. *IEEE Transactions on Robotics and Automation*, **10:1**, Feb. 1994.

[16] R. Simmons, R. Goodwin, K. Zita Haigh, S. Koenig and J. O'Sullivan. A Layered Architecture for Office Delivery Robots, Proc. Autonomous Agents '97, pp 245-252, Marina del Rey, CA, February 1997.