A Development Tool Environment for Configuration, Build, and Launch of Complex Applications

Mike Krueger
mmk@mc.com

Mercury Computer Systems, Inc.
199 Riverneck Road, Chelmsford, MA 01824

Abstract

The increasing size and complexity of high-performance applications have motivated a new round of innovation related to configuration, build, and launch of applications for large computing platforms, especially heterogeneous multicomputers. This paper describes the software technology of the Talaris™ Environment, created by Mercury Computer Systems, Inc. to enable a new generation of tools that construct and initiate applications for large distributed and parallel computer systems.

The Talaris Environment provides an extensible framework for cooperating tools that share application configuration information. Tools developed by Mercury for the Environment focus on high-performance embedded DSP applications that run on Mercury's RACE® series multicomputer systems. Additional tools under development by Mercury and other organizations support other target systems and programming interfaces that include UNIX workstation networks, the IBM SP/2, real-time DSP platforms, the Message Passing Interface(MPI), and POSIX.

Development of the Talaris Environment has been funded in part by the Defense Advanced Research Projects Agency (DARPA) under the "Bridging the Gap" and "Three Steps" programs. The Talaris Environment is currently available in connection with these DARPA programs.

1. Introduction

Software developers naturally focus on the feasibility and performance of the code that implements key algorithms. Today's integrated development environments have the same emphasis, offering powerful editing and browsing tools that focus on source-code development and debugging.

All developers understand they must also ensure the sources are compiled, linked, and packaged for successful execution on a target system. This work includes software integration, build, initialization, and execution. Typically these areas are pursued with a combination of program constants, makefiles, shell scripts, and the like. Software components are simply integrated by linking into a few executables that are executed on a single processor.

Though the engineering of this "glue" often takes more time than one might like, it is not usually considered a major factor in the development process. If available at all, the tool support for these activities takes the form of "project files" and "project settings" that define the population of software components and linking options.

For a *complex application*, however, component integration and execution problems are far from trivial, require a specific investment, and pose technical and schedule risks to the development project. For purposes of discussion in this paper, the nominal complex application comprises hundreds of software modules and distributed or parallel computer of similar scale.

1.1. Limits of the Conventional Approach

When conventional development tools and methods are used to create a complex application, very important configuration information is typically decentralized in both location and technology. Names of software communications mechanisms, ordinals indicating rank in groups of identical functions, processor assignment, order of startup, and many other related data are spread across header files, makefiles, initialization functions, startup sequencing functions, and shell scripts.

Another complicating factor for complex systems is that the hardware configuration involved in the application – that often differs across development and deployment phases – has a strong connection to the software configuration. When conventional tools fail to explicitly represent the hardware, critical software and hardware relationships are easily obscured.

1.2. The Talaris Environment

The fundamental assertion of this paper, and of the work that it describes, is that an environment for sharing centralized configuration information enables a new class of tools that configure, build, and launch complex applications. Mercury has created the Talaris Environment for exactly this purpose.

The main impact of the Environment is a beneficial change in the developer's workflow. Code relating to configuration and initialization of resources is moved into the Environment. The developer does far less engineering of configuration, build, and startup software, eliminating common errors in the process. Programs become more easily ported between target systems of differing scale, provided Talaris exists for both environments.

The Environment has three major subsystems:

1. The central *Talaris Core* that holds the application configuration data, allows access by multiple tools, and supports dynamic editing of a configuration document.

2. The *Talaris Editing Tools* with which the developer views and modifies the configuration data.

3. Interchangeable *Talaris Target Tools* that use configuration data as specification to build applications for launch on target systems.

The Environment is implemented in Java™ and runs on both Solaris™ 2 and Windows® NT 4.0 workstations. Tools can be readily configured in or out of the Environment, the behavior of the environment can be readily changed to suit application needs, and the type of target system can be switched.

2. The Talaris Core

The fundamental role of the Talaris Core is to provide the *model* in the model-view-controller (M-V-C) architecture of the Environment. The M-V-C architecture, a widely used approach to interactive software, allows tools to simultaneously share the model and present a consistent "live" view of the current state of the model.

The Core includes:

- The *Talaris Model*

- Support for the *Application Configuration Language (ACL)*

- The *Talaris Modeler* that manages the Model and ACL configuration documents

- The *Modeler Interface* through which tools access the Model

2.1. The Talaris Model

The key to the Environment is the centralized configuration information that its tools share via the Model.

Components

The most basic function of the Model is to represent software and hardware components and their relationships.

It is immediately apparent that an application's primary components are naturally organized into broad categories called *domains*. For example, a hand-drawn diagram of an application will typically depict software and hardware components in two separate diagrams. Components are related *within* and *across* domains in three ways.

- *Connections* define the means by which components within a domain can communicate with each other.

- *Groups* are collections of components within a domain. Development tools usually support groups in terms of subdiagrams, but this is a limited employment of the concept. A general approach to groups permits overlapping (multiple membership) and is *not* simply a means of organizing a visual representation of the configuration.

- *Assignments* across domains define the utilization of computing resources. A simple example is an assignment of a function in the software domain to a processor in the hardware domain.

Beyond Components: Application Objects

Software functions, processes, processors, and device controllers are obviously components in a high-level view of application software and hardware. For complex applications, however, it is also necessary to explicitly represent the means of communication between components. It is not sufficient to simply "draw a line" between components.

The Talaris Model addresses this issue with *ports* and *connections* as first-class objects in the configuration. Ports are associated with components and represent the communication behavior expected by the component. Connections represent the communication mechanism itself, which obviously must be compatible with all ports that it connects.

Taken together, components, ports, and connections are the *application objects* in the Model.

Overview of the Talaris Model

Complete coverage of Model semantics is well beyond the scope of this paper. In summary, the capabilities of the Model include:

- A *type hierarchy* of application object derived types, which can be edited dynamically

- *Domains* into which application types and objects are partitioned

- *Components* that are the primary application objects

- *Ports* associated with components to specify acceptable connection protocols

- *Connections* that explicitly represent means of communication between ports

- *Attributes* and *properties* that specify fixed or variable name/value data, applicable for all objects (including ports and connections)

- *Groups* that provide a general means of collecting objects

- *Assignment* relationships of objects across domains

- *Scaling* as a means of replication without multiple objects

- *Parameters* defining constants for the configuration

- *Expressions* using built-in *functions* and *operators*

- *Packages* that provide pre-defined application object types in separate namespaces

- *Templates* for parameterized instantiation of configuration fragments

- Single-statement iteration without looping constructs

2.2. The Application Configuration Language

It is reasonable to assume that the Talaris Environment would provide some sort of persistent representation of the information in the Model. This issue goes far beyond the obvious need to share and re-use configuration data, however.

Even if a configuration adheres to regular patterns of connections and assignments, a conventional graphical representation can be impenetrable and, for practical development purposes, unusable. The limitations of graphical representation strongly suggest that configuration information should be fully expressible in some sort of configuration programming language. This conclusion corresponds to industry experience with development tools for algorithm implementation, in which visual (i.e., pictorial) programming tools exist but programming languages dominate.

Consequently, the Environment includes support for the Application Configuration Language (ACL), a text command language that can completely specify a Model. The Environment goes one step further by supporting ACL as a full peer to any graphical user interfaces. This provides an unusual and very powerful feature: the dynamic editing of a *configuration document* as a free side effect of the actions of all Talaris Tools. This capability is central to the Environment, and the Environment's main window is a dynamic display of the current configuration document.

Overview of ACL

The following partial list of ACL commands does not describe the complete syntax of ACL, but the general purpose of the commands should be clear.

declare *base-type new-type*
    derive a new type from an existing base type

delete *type*
    remove a type from the type hierarchy

parameter *name expression*
    define a new parameter

create *type name<scale> properties*
    create a new object (optionally scaled)

delete *object*
    delete an application object

set_scale *component-name<new-scale>*
    change the scale of an object

set_property *object property value*
    add/change a property for an object

connect *connection component-1.port component-2.port*
    connect ports with a connection

assign *object-1 -to object-2*
    assign (map) objects in different domains

group *lead-object member-objects...*
    associate objects in a group

apply *{ var-1 values-1 } { var-2 values-2 } ... command-using-vars*
    iterate variables, executing command

package *name { param-1 value-1 } { param-2 value-2} ...*
    designate ACL document as reusable package with default parameter values

use *package-name { param-1 arg-1 } { param-2 arg-2} ...*
    invoke package with arguments for parameters

template *name { param-1 value-1 } { param-2 value-2 } ...*
    designate ACL document as parameterized ACL document

instantiate *name namespace { param-1 value-1 } { param-2 value-2} ...*
    invoke template with arguments for parameters, qualified by namespace

2.3. The Talaris Modeler

The Model is controlled by a transaction engine called the *Talaris Modeler* that manages and synchronizes (in real-time) both the Model *and* the corresponding ACL command source. The Modeler is equally at ease with the ACL source form of the Model and the underlying Model itself, with its computed entities and relationships. Because the Modeler supports a Model-View-Controller architecture, all views of the Model – including the ACL command form – are constantly synchronized as the contents of the Model are changed. This approach has been used to enable some powerful features:

- Developers at ease with the ACL language can use text-oriented tools to browse and edit the source text, and instantly see the effects of those changes to the Model in graphical views provided by other tools.

- Developers more comfortable with manipulating the Model in terms of its entities and relationships can do so with GUI tools, and instantly see how those changes are optimally expressed in the ACL command form.

- Values in ACL can be expressed either as literal values or as expressions using parameters. When perusing the contents of the Model, the developer has the choice of viewing the values either as evaluated or in the form of the original expression.

- Unlimited undo of editing actions on the Model.

To achieve the effect of synchronizing the Model with an ACL document, the Model Transaction Engine includes the following mechanisms:

- Backwriting the Model in ACL form to the current configuration document

- Optimization of ACL command content and ordering

- Reflection of dependencies on deleted entities

- Roll-back/roll-forward of transactions according to command dependencies

- Full source correspondence for expressions

2.4. The Modeler Interface

Tools in the Environment use the Modeler Interface (also called the Modeler API) to gain full access to the Model. The Modeler API is organized into the following general categories: Query, Modification, and Observation.

The Modeler Query API

Tools can query the Modeler about any aspect of the current Model. Inventories of the current types and instances, properties of a given type or instance, relationship information about a given instance or port, and the current set of Model Parameters and their values can all be obtained through the Modeler API.

A particularly powerful feature of the query API is the use of *object references*. Nearly all queries are formed by first building an object which describes the entity to be queried. This could be, for example, a type, an instance, a single member of a scaled instance, a port, or a single member of a scaled port. This enables a small, concise query API to handle a broad variety of query types. Queries that reference non-existent entities, or make nonsensical queries of an existent entity, are rejected using the Java runtime exception mechanism.

The Modeler Modification API

Tools can change any aspect of the current Model. Types and instances can be created and deleted, properties on those entities can be modified and deleted, types and instances can be rescaled, Model Parameters can be changed, and relationships between entities can be created and deleted. Any change that can be effected via an atomic ACL command (i.e., a non-third-party ACL command) has an equivalent API call in the Modeler. Like the Query API, modifications are done via object references, and requests that fail are rejected using exceptions.

The Modeler Observation API

Any Java object can register with the Modeler to receive change notifications. There is a large number of change notifications, such as entity lifetime (creation/deletion), changes to properties on entities, changes in relationships in the Model, and changes to Transactions. A given Java object can register for only the types of changes it is interested in observing, and not be bothered with the remaining changes. The Model-View-Controller architecture employed by the Modeler results in the following powerful features for the environment:

- Tools that initiate changes to the Model need not infer the actual effect of the change. Rather, the tool can assume the change succeeded if the Modeler does not throw an exception. The changes to the Model will be announced in detail as part of the Observation API.

- A tool does not need to be the initiator of a change to see the effect of that change. This allows the effect of a change to be displayed by multiple tools that are not formally integrated with the tool that initiates the change.

3. Talaris Tools

On startup, the Environment uses initialization settings to establish the semantics of the Model, the number and names of Model domains, the initial type hierarchy, and the selected target system. The developer proceeds to open, edit, and save ACL documents with Editing Tools. When the configuration is complete, the developer uses Target Tools to build and launch the application on a specific target system.

3.1. Editing Tools

In the Environment, the term "editing tool" is used in the broad sense of any tool that helps formulate, revise, or analyze the Model. Editing tools are readily added to the environment so they appear on the standard toolbar and menus of other tools.

Mercury has developed a suite of seven basic editing tools that provides complete coverage of all possible operations on the Model. These tools use hierarchical lists and tables for visual presentation and support many drag-and-drop gestures. By foregoing pictorial (i.e., box/bubble/line) graphical user interfaces, these tools provide relatively compact displays of Model information.

Mercury's editing tools include:

- The Document Tool – the "main window" for the environment, performing open/save/close and dynamic display of the current configuration document. Referenced ACL documents (such as packages or templates) appear in-line as a hierarchical expansion of the ACL document display.

- The Inspector – a browser/editor for application object types and instances.

- The Mapper – edits assignment relationships.

- The Connector and the Grouper – edits connection and group relationships.

- The Relation Tool – summarizes object relationships of all kinds.

- The Parameter Tool – edits parameter definitions.

Mercury has also created a "Tcl Tool" that supports access of the model via interactive or scripted Tcl commands. This tool has proved to be useful for testing and for access by external development tools that are not fully integrated with the environment.

3.2. Target Tools

Talaris Target Tools build applications for target systems and, in most cases, also load, initialize, and start the application. The RACE Target Package has been created by Mercury for its MC/OS™ runtime environment. In support of the BAA97-06 "Three Steps" project, Mercury is currently developing a UNIX Target Package, initially targeted for Solaris 2, with support for POSIX and MPI 1.2 application programming interfaces. Mercury is concurrently adding MPI 1.2 support to the RACE Target Package. Also, for the "Three Steps" project, the UNIX Target Package will be extended to support the SP/2 by MHPCC.

Mercury's RACE Generator Tool

To create applications to launch on RACE systems, Mercury's RACE Generator Tool builds a *launch kit* in a sequence of four phases:

- Analysis – scan the Model, resolve ambiguities and other defaults, perform initialization deadlock analysis

- Files – create all the input to the launch kit, initialization instructions, dependency file, dispatch tables

- Build – compile dispatch tables, build the executable images

- Kit – create a launch kit from the output of the files and build phases

To reduce the time required to create a launch kit, the Generator computes the minimum build plan. The analysis and files phases are done only if the model has changed, the build phase is done only if its inputs have changed, and the kit phase updates the final kit contents only if the build phase resulted in changes. This enables fast turnaround during kit generation as well as during launch, because redundant image-loads are thereby avoided.

Mercury's RACE Launcher

Mercury's RACE Launcher starts applications on RACE systems in four steps:

- **Image load.** Note that a sizeable application may involve several hundred megabytes of executable images being loaded onto hundreds of processors. Using features of MC/OS, the RACE Launcher avoids reloading images that have not changed since the last launch.

- **Process and thread creation.** The Launcher constructs argc/argv/env data for processes and marshals arguments for thread entry points.

- **Initialization.** To start properly and without deadlock, a complex application requires precise setup of thousands of software communication mechanisms. The Launcher is able to do this in coordinated phases, avoiding possible deadlock conditions in a highly orchestrated communication protocol between the Launcher and small *agent modules* that have been placed in each process by the Generator.

- **Initiation.** With all initialization complete, the Launcher releases threads and starts processes.

4. Summary

Mercury has created a powerful environment for cooperating tools that better support the development of complex applications. Implemented in Java, the Talaris Environment is very portable and extensible. Today, it is being applied to an increasing range of target systems and programming interfaces. Mercury, with other companies, is working to make full application portability a reality for complex applications. For example, current work will result in support for the MPI programming interface on both RACE and Solaris target systems. MPI-based applications configured and built with the Environment will be highly portable.

Building on today's basic tool set, future Talaris tools in development or under consideration include application-specific "wizards" that compute application configurations according to data shape, performance prediction simulators, automatic software/hardware mapping tools, and performance data presentation tools.

Beyond tool additions, future opportunities for the Talaris Environment include the likely use of the Environment as a platform for very different purposes. As thoroughly as tools exercise the Talaris Core today, the potential of this dynamic modeling substrate is yet to be fully understood and explored.