

```

*****
94913 Wed May 20 12:01:44 2015
new/usr/src/cmd/make/bin/doname.cc
make: restore a couple of blocks of code from DISTRIBUTED that should have been
*****
_____unchanged_portion_omitted_____

901 /*
902 * DONE.
903 *
904 * check_dependencies(result, line, do_get,
905 * target, true_target, doing_subtree, out_of_date_tail,
906 * old_locals, implicit, command, less, rechecking_target)
907 *
908 * Return value:
909 * True returned if some dependencies left running
910 *
911 * Parameters:
912 * result Pointer to cell we update if build failed
913 * line We get the dependencies from here
914 * do_get Allow use of sccs get in recursive doname()
915 * target The target to chase dependencies for
916 * true_target The real one for :: and lib(member)
917 * doing_subtree True if building a conditional macro subtree
918 * out_of_date_tail Used to set the $? list
919 * old_locals Used for resetting the local macros
920 * implicit Called when scanning for implicit rules?
921 * command Place to stuff command
922 * less Set to $< value
923 *
924 * Global variables used:
925 * command_changed Set if we suspect .make.state needs rewrite
926 * debug_level Should we trace actions?
927 * force The Name " FORCE", compared against
928 * recursion_level Used for tracing
929 * rewrite_statefile Set if .make.state needs rewriting
930 * wait_name The Name ".WAIT", compared against
931 */
932 static Boolean
933 check_dependencies(Doname *result, Property line, Boolean do_get, Name target, N
934 {
935 Boolean dependencies_running;
936 register Dependency dependency;
937 Doname dep_result;
938 Boolean dependency_changed = false;

940 line->body.line.dependency_time = file_doesnt_exist;
941 if (line->body.line.query != NULL) {
942 delete_query_chain(line->body.line.query);
943 }
944 line->body.line.query = NULL;
945 line->body.line.is_out_of_date = false;
946 dependencies_running = false;
947 /*
948 * Run thru all the dependencies and call doname() recursively
949 * on each of them.
950 */
951 for (dependency = line->body.line.dependencies;
952 dependency != NULL;
953 dependency = dependency->next) {
954 Boolean this_dependency_changed = false;

956 if (!dependency->automatic &&
957 (rechecking_target || target->rechecking_target)) {
958 /*
959 * We only bother with the autos when rechecking

```

```

960 */
961 continue;
962 }

964 if (dependency->name == wait_name) {
965 /*
966 * The special target .WAIT means finish all of
967 * the prior dependencies before continuing.
968 */
969 if (dependencies_running) {
970 break;
971 }
972 } else if ((!parallel_ok(dependency->name, false)) &&
973 (dependencies_running)) {
974 /*
975 * If we can't execute the current dependency in
976 * parallel, hold off the dependency processing
977 * to preserve the order of the dependencies.
978 */
979 break;
980 #endif /* ! codereview */
981 } else {
982 timestruc_t depe_time = file_doesnt_exist;

985 if (true_target->is_member) {
986 depe_time = exists(dependency->name);
987 }
988 if (dependency->built ||
989 (dependency->name->state == build_failed)) {
990 dep_result = (Doname) dependency->name->state;
991 } else {
992 dep_result = doname_check(dependency->name,
993 do_get,
994 false,
995 (Boolean) dependency->
996 }
997 if (true_target->is_member || dependency->name->is_membe
998 /* should compare only secs, cause lib members d
999 if (depe_time.tv_sec != dependency->name->stat.t
1000 this_dependency_changed =
1001 dependency_changed =
1002 true;
1003 }
1004 } else {
1005 if (depe_time != dependency->name->stat.time) {
1006 this_dependency_changed =
1007 dependency_changed =
1008 true;
1009 }
1010 }
1011 dependency->built = true;
1012 switch (dep_result) {
1013 case build_running:
1014 dependencies_running = true;
1015 continue;
1016 case build_failed:
1017 *result = build_failed;
1018 break;
1019 case build_dont_know:
1020 /*
1021 * If make can't figure out how to make a dependency, maybe the dependency
1022 * is out of date. In this case, we just declare the target out of date
1023 * and go on. If we really need the dependency, the make'ing of the target
1024 * will fail. This will only happen for automatic (hidden) dependencies.
1025 */

```

```

1026     if(!reread_conditionals) {
1027         line->body.line.is_out_of_date = true;
1028     }
1029     /*
1030     * Make sure the dependency is not saved
1031     * in the state file.
1032     */
1033     dependency->stale = true;
1034     rewrite_statefile =
1035     command_changed =
1036     true;
1037     if (debug_level > 0) {
1038         (void) printf(catgets(catd, 1, 19, "Targ
1039         true_target->string_mb,
1040         dependency->name->string_mb
1041     )
1042     break;
1043     }
1044     if (dependency->name->depends_on_conditional) {
1045         target->depends_on_conditional = true;
1046     }
1047     if (dependency->name == force) {
1048         target->stat.time =
1049         dependency->name->stat.time;
1050     }
1051     /*
1052     * Propagate new timestamp from "member" to
1053     * "lib.a(member)".
1054     */
1055     (void) exists(dependency->name);

1057     /* Collect the timestamp of the youngest dependency */
1058     line->body.line.dependency_time =
1059     MAX(dependency->name->stat.time,
1060     line->body.line.dependency_time);

1062     /* Correction: do not consider nanosecs for members */
1063     if(true_target->is_member || dependency->name->is_member)
1064         line->body.line.dependency_time.tv_nsec = 0;
1065     }

1067     if (debug_level > 1) {
1068         (void) printf(catgets(catd, 1, 20, "%*sDate(%s)=
1069         recursion_level,
1070         "",
1071         dependency->name->string_mb,
1072         time_to_string(dependency->name->
1073         stat.time));
1074         if (dependency->name->stat.time > line->body.lin
1075         (void) printf(catgets(catd, 1, 21, "%*sD
1076         recursion_level,
1077         "",
1078         true_target->string_mb,
1079         time_to_string(line->body.
1080         dependency_
1081         )
1082     }

1084     /* Build the $? list */
1085     if (true_target->is_member) {
1086         if (this_dependency_changed == true) {
1087             true_target->stat.time = dependency->nam
1088             true_target->stat.time.tv_sec--;
1089         } else {
1090             /* Dina:
1091             * The next statement is commented

```

```

1092     * out as a fix for bug #1051032.
1093     * if dependency hasn't changed
1094     * then there's no need to invalidate
1095     * true_target. This statemnt causes
1096     * make to take much longer to process
1097     * an already-built archive. Soren
1098     * said it was a quick fix for some
1099     * problem he doesn't remember.
1100     true_target->stat.time = file_no_time;
1101     */
1102     (void) exists(true_target);
1103     }
1104     } else {
1105         (void) exists(true_target);
1106     }
1107     Boolean out_of_date;
1108     if (true_target->is_member || dependency->name->is_membe
1109         out_of_date = (Boolean) OUT_OF_DATE_SEC(true_tar
1110         depend
1111     )
1112     } else {
1113         out_of_date = (Boolean) OUT_OF_DATE(true_target-
1114         dependency->
1115     )
1116     if ((build_unconditional || out_of_date) &&
1117         (dependency->name != force) &&
1118         (dependency->stale == false)) {
1119         *out_of_date_tail = ALLOC(Chain);
1120         if (dependency->name->is_member &&
1121             (get_prop(dependency->name->prop,
1122             member_prop) != NULL)) {
1123             (*out_of_date_tail)->name =
1124             get_prop(dependency->name->prop,
1125             member_prop)->
1126             body.member.member;
1127         } else {
1128             (*out_of_date_tail)->name =
1129             dependency->name;
1130         }
1131         (*out_of_date_tail)->next = NULL;
1132         out_of_date_tail = &(*out_of_date_tail)->next;
1133         if (debug_level > 0) {
1134             if (dependency->name->stat.time == file_
1135                 (void) printf(catgets(catd, 1, 2
1136                 recursion_level,
1137                 "",
1138                 true_target->strin
1139                 dependency->name->
1140             )
1141             } else {
1142                 (void) printf(catgets(catd, 1, 2
1143                 recursion_level,
1144                 "",
1145                 true_target->strin
1146                 dependency->name->
1147             )
1148         }
1149         if (dependency->name == force) {
1150             force->stat.time =
1151             file_max_time;
1152             force->state = build_dont_know;
1153         }
1154     }
1155     if (dependencies_running) {
1156         if (doing_subtree) {
1157             if (target->conditional_cnt > 0) {

```

```

1158         reset_locals(target,
1159                       old_locals,
1160                       get_prop(target->prop,
1161                               conditional_prop),
1162                       0);
1163     }
1164     return true;
1165 } else {
1166     target->state = build_running;
1167     add_pending(target,
1168               --recursion_level,
1169               do_get,
1170               implicit,
1171               false);
1172     if (target->conditional_cnt > 0) {
1173         reset_locals(target,
1174                   old_locals,
1175                   get_prop(target->prop,
1176                           conditional_prop),
1177                   0);
1178     }
1179     return true;
1180 }
1181 }
1182 /*
1183  * Collect the timestamp of the youngest double colon target
1184  * dependency.
1185  */
1186 if (target->is_double_colon_parent) {
1187     for (dependency = line->body.line.dependencies;
1188          dependency != NULL;
1189          dependency = dependency->next) {
1190         Property tmp_line;
1191
1192         if ((tmp_line = get_prop(dependency->name->prop, line_pr
1193                                 if(tmp_line->body.line.dependency_time != file_m
1194                                     target->stat.time =
1195                                         MAX(tmp_line->body.line.dependency_tim
1196                                             target->stat.time);
1197     }
1198 }
1199 }
1200 }
1201 if ((true_target->is_member) && (dependency_changed == true)) {
1202     true_target->stat.time = file_no_time;
1203 }
1204 /*
1205  * After scanning all the dependencies, we check the rule
1206  * if we found one.
1207  */
1208 if (line->body.line.command_template != NULL) {
1209     if (line->body.line.command_template_redefined) {
1210         warning(catgets(catd, 1, 24, "Too many rules defined for
1211                 target->string_mb);
1212     }
1213     *command = line;
1214     /* Check if the target is out of date */
1215     Boolean out_of_date;
1216     if (true_target->is_member) {
1217         out_of_date = (Boolean) OUT_OF_DATE_SEC(true_target->sta
1218                                             line->body.line.
1219     } else {
1220         out_of_date = (Boolean) OUT_OF_DATE(true_target->stat.ti
1221                                             line->body.line.depe
1222     }
1223     if (build_unconditional || out_of_date){

```

```

1224         if(!rerecheck_conditionals) {
1225             line->body.line.is_out_of_date = true;
1226         }
1227     }
1228     line->body.line.sccs_command = false;
1229     line->body.line.target = true_target;
1230     if(gnu_style) {
1231
1232         // set $< for explicit rule
1233         if(line->body.line.dependencies != NULL) {
1234             less = line->body.line.dependencies->name;
1235         }
1236
1237         // set $* for explicit rule
1238         Name target_body;
1239         Name tt = true_target;
1240         Property member;
1241         register wchar_t *target_end;
1242         register Dependency suffix;
1243         register int suffix_length;
1244         Wstring targ_string;
1245         Wstring suf_string;
1246
1247         if (true_target->is_member &&
1248             ((member = get_prop(target->prop, member_prop)) !=
1249              NULL)) {
1250             tt = member->body.member.member;
1251         }
1252         targ_string.init(tt);
1253         target_end = targ_string.get_string() + tt->hash.length;
1254         for (suffix = suffixes; suffix != NULL; suffix = suffix-
1255             suffix_length = suffix->name->hash.length;
1256             suf_string.init(suffix->name);
1257             if (tt->hash.length < suffix_length) {
1258                 continue;
1259             } else if (!IS_EQUALN(suf_string.get_string(),
1260                                 (target_end - suffix_length),
1261                                 suffix_length)) {
1262                 continue;
1263             }
1264             target_body = GETNAME(
1265                 targ_string.get_string(),
1266                 (int)(tt->hash.length - suffix_length)
1267             );
1268             line->body.line.star = target_body;
1269         }
1270
1271         // set result = build_ok so that implicit rules are not
1272         if(*result == build_dont_know) {
1273             *result = build_ok;
1274         }
1275     }
1276     if (less != NULL) {
1277         line->body.line.less = less;
1278     }
1279 }
1280
1281 return false;
1282 }
1283
1284 /*
1285  * dynamic_dependencies(target)
1286  *
1287  * Checks if any dependency contains a macro ref
1288  * If so, it replaces the dependency with the expanded version.
1289  * Here, "$@" gets translated to target->string. That is

```

```

1290 * the current name on the left of the colon in the
1291 * makefile. Thus,
1292 * xyz: s.$@.c
1293 * translates into
1294 * xyz: s.xyz.c
1295 *
1296 * Also, "$(@F)" translates to the same thing without a preceding
1297 * directory path (if one exists).
1298 * Note, to enter "$@" on a dependency line in a makefile
1299 * "$$$@" must be typed. This is because make expands
1300 * macros in dependency lists upon reading them.
1301 * dynamic_dependencies() also expands file wildcards.
1302 * If there are any Shell meta characters in the name,
1303 * search the directory, and replace the dependency
1304 * with the set of files the pattern matches
1305 *
1306 * Parameters:
1307 * target Target to sanitize dependencies for
1308 *
1309 * Global variables used:
1310 * c_at The Name "@", used to set macro value
1311 * debug_level Should we trace actions?
1312 * dot The Name ".", used to read directory
1313 * recursion_level Used for tracing
1314 */
1315 void
1316 dynamic_dependencies(Name target)
1317 {
1318     wchar_t pattern[MAXPATHLEN];
1319     register wchar_t *p;
1320     Property line;
1321     register Dependency dependency;
1322     register Dependency *remove;
1323     String_rec string;
1324     wchar_t buffer[MAXPATHLEN];
1325     register Boolean set_at = false;
1326     register wchar_t *start;
1327     Dependency new_depe;
1328     register Boolean reuse_cell;
1329     Dependency first_member;
1330     Name directory;
1331     Name lib;
1332     Name member;
1333     Property prop;
1334     Name true_target = target;
1335     wchar_t *library;

1337     if ((line = get_prop(target->prop, line_prop)) == NULL) {
1338         return;
1339     }
1340     /* If the target is constructed from a "::" target we consider that */
1341     if (target->has_target_prop) {
1342         true_target = get_prop(target->prop,
1343             target_prop->body.target.target;
1344     }
1345     /* Scan all dependencies and process the ones that contain "$" chars */
1346     for (dependency = line->body.line.dependencies;
1347         dependency != NULL;
1348         dependency = dependency->next) {
1349         if (!dependency->name->dollar) {
1350             continue;
1351         }
1352         target->has_depe_list_expanded = true;

1354     /* The make macro $@ is bound to the target name once per */
1355     /* invocation of dynamic_dependencies() */

```

```

1356     if (!set_at) {
1357         (void) SETVAR(c_at, true_target, false);
1358         set_at = true;
1359     }
1360     /* Expand this dependency string */
1361     INIT_STRING_FROM_STACK(string, buffer);
1362     expand_value(dependency->name, &string, false);
1363     /* Scan the expanded string. It could contain whitespace */
1364     /* which mean it expands to several dependencies */
1365     start = string.buffer.start;
1366     while (iswspace(*start)) {
1367         start++;
1368     }
1369     /* Remove the cell (later) if the macro was empty */
1370     if (start[0] == (int) nul_char) {
1371         dependency->name = NULL;
1372     }

1374 /* azv 10/26/95 to fix bug BID_1170218 */
1375     if ((start[0] == (int) period_char) &&
1376         (start[1] == (int) slash_char)) {
1377         start += 2;
1378     }
1379 /* azv */

1381     first_member = NULL;
1382     /* We use the original dependency cell for the first */
1383     /* dependency from the expansion */
1384     reuse_cell = true;
1385     /* We also have to deal with dependencies that expand to */
1386     /* lib.a(members) notation */
1387     for (p = start; *p != (int) nul_char; p++) {
1388         if ((*p == (int) parenleft_char)) {
1389             lib = GETNAME(start, p - start);
1390             lib->is_member = true;
1391             first_member = dependency;
1392             start = p + 1;
1393             while (iswspace(*start)) {
1394                 start++;
1395             }
1396             break;
1397         }
1398     }
1399     do {
1400         /* First skip whitespace */
1401         for (p = start; *p != (int) nul_char; p++) {
1402             if ((*p == (int) nul_char) ||
1403                 iswspace(*p) ||
1404                 (*p == (int) parenright_char)) {
1405                 break;
1406             }
1407         }
1408         /* Enter dependency from expansion */
1409         if (p != start) {
1410             /* Create new dependency cell if */
1411             /* this is not the first dependency */
1412             /* picked from the expansion */
1413             if (!reuse_cell) {
1414                 new_depe = ALLOC(Dependency);
1415                 new_depe->next = dependency->next;
1416                 new_depe->automatic = false;
1417                 new_depe->stale = false;
1418                 new_depe->built = false;
1419                 dependency->next = new_depe;
1420                 dependency = new_depe;
1421             }

```

```

1422 reuse_cell = false;
1423 /* Internalize the dependency name */
1424 // tolik. Fix for bug 4110429: inconsistent expa
1425 // include "/" and "/"
1426 //dependency->name = GETNAME(start, p - start);
1427 dependency->name = normalize_name(start, p - sta
1428 if ((debug_level > 0) &&
1429 (first_member == NULL)) {
1430 (void) printf(catgets(catd, 1, 25, "%*sD
1431 recursion_level,
1432 "",
1433 dependency->name->string_m
1434 true_target->string_mb);
1435 }
1436 for (start = p; iswspace(*start); start++);
1437 p = start;
1438 }
1439 } while ((*p != (int) nul_char) &&
1440 (*p != (int) parenright_char));
1441 /* If the expansion was of lib.a(members) format we now */
1442 /* enter the proper member cells */
1443 if (first_member != NULL) {
1444 /* Scan the new dependencies and transform them from */
1445 /* "foo" to "lib.a(foo)" */
1446 for (; 1; first_member = first_member->next) {
1447 /* Build "lib.a(foo)" name */
1448 INIT_STRING_FROM_STACK(string, buffer);
1449 APPEND_NAME(lib,
1450 &string,
1451 (int) lib->hash.length);
1452 append_char((int) parenleft_char, &string);
1453 APPEND_NAME(first_member->name,
1454 &string,
1455 FIND_LENGTH);
1456 append_char((int) parenright_char, &string);
1457 member = first_member->name;
1458 /* Replace "foo" with "lib.a(foo)" */
1459 first_member->name =
1460 GETNAME(string.buffer.start, FIND_LENGTH);
1461 if (string.free_after_use) {
1462 retmem(string.buffer.start);
1463 }
1464 if (debug_level > 0) {
1465 (void) printf(catgets(catd, 1, 26, "%*sD
1466 recursion_level,
1467 "",
1468 first_member->name->
1469 string_mb,
1470 true_target->string_mb);
1471 }
1472 first_member->name->is_member = lib->is_member;
1473 /* Add member property to member */
1474 prop = maybe_append_prop(first_member->name,
1475 member_prop);
1476 prop->body.member.library = lib;
1477 prop->body.member.entry = NULL;
1478 prop->body.member.member = member;
1479 if (first_member == dependency) {
1480 break;
1481 }
1482 }
1483 }
1484 }
1485 Wstring wcb;
1486 /* Then scan all the dependencies again. This time we want to expand */
1487 /* shell file wildcards */

```

```

1488 for (remove = &line->body.line.dependencies, dependency = *remove;
1489 dependency != NULL;
1490 dependency = *remove) {
1491 if (dependency->name == NULL) {
1492 dependency = *remove = (*remove)->next;
1493 continue;
1494 }
1495 /* If dependency name string contains shell wildcards */
1496 /* replace the name with the expansion */
1497 if (dependency->name->wildcard) {
1498 wcb.init(dependency->name);
1499 if ((start = (wchar_t *) wschr(wcb.get_string(),
1500 (int) parenleft_char)) != NULL) {
1501 /* lib(*) type pattern */
1502 library = buffer;
1503 (void) wncpy(buffer,
1504 wcb.get_string(),
1505 start - wcb.get_string());
1506 buffer[start-wcb.get_string()] =
1507 (int) nul_char;
1508 (void) wncpy(pattern,
1509 start + 1,
1510 (int) (dependency->name->hash.length-(start-wcb.get_string()-2));
1511 pattern[dependency->name->hash.length -
1512 (start-wcb.get_string() - 2)] =
1513 (int) nul_char;
1514 } else {
1515 library = NULL;
1516 (void) wncpy(pattern,
1517 wcb.get_string(),
1518 (int) dependency->name->hash.length);
1519 pattern[dependency->name->hash.length] =
1520 (int) nul_char;
1521 }
1522 start = (wchar_t *) wschr(pattern, (int) slash_char);
1523 if (start == NULL) {
1524 directory = dot;
1525 p = pattern;
1526 } else {
1527 directory = GETNAME(pattern, start-pattern);
1528 p = start+1;
1529 }
1530 /* The expansion is handled by the read_dir() routine*/
1531 if (read_dir(directory, p, line, library)) {
1532 *remove = (*remove)->next;
1533 } else {
1534 remove = &dependency->next;
1535 }
1536 } else {
1537 remove = &dependency->next;
1538 }
1539 }
1540 }
1541 /* Then unbind $@ */
1542 (void) SETVAR(c_at, (Name) NULL, false);
1543 }
1544 */
1545 * DONE.
1546 *
1547 *
1548 * run_command(line)
1549 *
1550 * Takes one Cmd_line and runs the commands from it.
1551 *
1552 * Return value:
1553 * Indicates if the command failed or not

```

```

1554 *
1555 *   Parameters:
1556 *       line           The command line to run
1557 *
1558 *   Global variables used:
1559 *       commands_done Set if we do run command
1560 *       current_line  Set to the line we run a command from
1561 *       current_target Set to the target we run a command for
1562 *       file_number   Used to form temp file name
1563 *       keep_state    Indicates that .KEEP_STATE is on
1564 *       make_state    The Name ".make.state", used to check timestamp
1565 *       parallel      True if currently building in parallel
1566 *       parallel_process_cnt Count of parallel processes running
1567 *       quest         Indicates that make -q is on
1568 *       rewrite_statefile Set if we do run a command
1569 *       sunpro_dependencies The Name "SUNPRO_DEPENDENCIES", set value
1570 *       temp_file_directory Used to form temp file name
1571 *       temp_file_name Set to the name of the temp file
1572 *       touch         Indicates that make -t is on
1573 */
1574 static Doname
1575 run_command(register Property line, Boolean)
1576 {
1577     register Doname      result = build_ok;
1578     register Boolean     remember_only = false;
1579     register Name       target = line->body.line.target;
1580     wchar_t             *string;
1581     char                tmp_file_path[MAXPATHLEN];
1582
1583     if (!line->body.line.is_out_of_date && target->rechecking_target) {
1584         target->rechecking_target = false;
1585         return build_ok;
1586     }
1587
1588     /*
1589     * Build the command if we know the target is out of date,
1590     * or if we want to check cmd consistency.
1591     */
1592     if (line->body.line.is_out_of_date || keep_state) {
1593         /* Hack for handling conditional macros in DMake. */
1594         if (!line->body.line.dont_rebuild_command_used) {
1595             build_command_strings(target, line);
1596         }
1597     }
1598     /* Never mind */
1599     if (!line->body.line.is_out_of_date) {
1600         return build_ok;
1601     }
1602     /* If quest, then exit(1) because the target is out of date */
1603     if (quest) {
1604         if (posix) {
1605             result = execute_parallel(line, true);
1606         }
1607         exit_status = 1;
1608         exit(1);
1609     }
1610     /* We actually had to do something this time */
1611     rewrite_statefile = commands_done = true;
1612     /*
1613     * If this is an sccs command, we have to do some extra checking
1614     * and possibly complain. If the file can't be gotten because it's
1615     * checked out, we complain and behave as if the command was
1616     * executed even though we ignored the command.
1617     */
1618     if (!touch &&
1619         line->body.line.sccs_command &&

```

```

1620         (target->stat.time != file_doesnt_exist) &&
1621         ((target->stat.mode & 0222) != 0)) {
1622             fatal(catgets(catd, 1, 27, "%s is writable so it cannot be sccs
1623                 target->string_mb);
1624             target->has_complained = remember_only = true;
1625         }
1626     /*
1627     * If KEEP_STATE is on, we make sure we have the timestamp for
1628     * .make.state. If .make.state changes during the command run,
1629     * we reread .make.state after the command. We also setup the
1630     * environment variable that asks utilities to report dependencies.
1631     */
1632     if (!touch &&
1633         keep_state &&
1634         !remember_only) {
1635         (void) exists(make_state);
1636         if ((strlen(temp_file_directory) == 1) &&
1637             (temp_file_directory[0] == '/')) {
1638             tmp_file_path[0] = '\0';
1639         } else {
1640             strcpy(tmp_file_path, temp_file_directory);
1641         }
1642         sprintf(mbs_buffer,
1643             NOCATGETS("%s/.make.dependency.%08x.%d.%d"),
1644             tmp_file_path,
1645             hostid,
1646             getpid(),
1647             file_number++);
1648         MBSTOWCS(wcs_buffer, mbs_buffer);
1649         Boolean fnd;
1650         temp_file_name = getname_fn(wcs_buffer, FIND_LENGTH, false, &fnd);
1651         temp_file_name->stat.is_file = true;
1652         int len = 2*MAXPATHLEN + strlen(target->string_mb) + 2;
1653         wchar_t *to = string = ALLOC_WC(len);
1654         for (wchar_t *from = wcs_buffer; *from != (int) nul_char; ) {
1655             if (*from == (int) space_char) {
1656                 *to++ = (int) backslash_char;
1657             }
1658             *to++ = *from++;
1659         }
1660         *to++ = (int) space_char;
1661         MBSTOWCS(to, target->string_mb);
1662         Name sprodep_name = getname_fn(string, FIND_LENGTH, false, &fnd);
1663         (void) SETVAR(sunpro_dependencies,
1664             sprodep_name,
1665             false);
1666         retmem(string);
1667     } else {
1668         temp_file_name = NULL;
1669     }
1670
1671     /*
1672     * In case we are interrupted, we need to know what was going on.
1673     */
1674     current_target = target;
1675     /*
1676     * We also need to be able to save an empty command instead of the
1677     * interrupted one in .make.state.
1678     */
1679     current_line = line;
1680     if (remember_only) {
1681         /* Empty block!!! */
1682     } else if (touch) {
1683         result = touch_command(line, target, result);
1684         if (posix) {
1685             result = execute_parallel(line, true);

```

```

1686     }
1687   } else {
1688     /*
1689     * If this is not a touch run, we need to execute the
1690     * proper command(s) for the target.
1691     */
1692     if (parallel) {
1693       if (!parallel_ok(target, true)) {
1694         /*
1695         * We are building in parallel, but
1696         * this target must be built in serial.
1697         */
1698         /*
1699         * If nothing else is building,
1700         * do this one, else wait.
1701         */
1702         if (parallel_process_cnt == 0) {
1703           result = execute_parallel(line, true, ta
1704         } else {
1705           current_target = NULL;
1706           current_line = NULL;
1707         /*
1708           line->body.line.command_used = NULL;
1709         */
1710           line->body.line.dont_rebuild_command_use
1711           return build_serial;
1712         }
1713       } else {
1714         result = execute_parallel(line, false);
1715         switch (result) {
1716         case build_running:
1717           return build_running;
1718         case build_serial:
1719           if (parallel_process_cnt == 0) {
1720             result = execute_parallel(line,
1721           } else {
1722             current_target = NULL;
1723             current_line = NULL;
1724             target->parallel = false;
1725             line->body.line.command_used =
1726             NULL;
1727             return build_serial;
1728           }
1729         }
1730       }
1731     } else {
1732       result = execute_parallel(line, true, target->localhost)
1733     }
1734   }
1735   temp_file_name = NULL;
1736   if (report_dependencies_level == 0){
1737     update_target(line, result);
1738   }
1739   current_target = NULL;
1740   current_line = NULL;
1741   return result;
1742 }

1744 /*
1745 * execute_serial(line)
1746 *
1747 * Runs thru the command line for the target and
1748 * executes the rules one by one.
1749 *
1750 * Return value:
1751 * The result of the command build

```

```

1752 *
1753 * Parameters:
1754 * line The command to execute
1755 *
1756 * Static variables used:
1757 *
1758 * Global variables used:
1759 * continue_after_error -k flag
1760 * do_not_exec_rule -n flag
1761 * report_dependencies -P flag
1762 * silent Don't echo commands before executing
1763 * temp_file_name Temp file for auto dependencies
1764 * vpath_defined If true, translate path for command
1765 */
1766 Doname
1767 execute_serial(Property line)
1768 {
1769   int child_pid = 0;
1770   Boolean printed_serial;
1771   Doname result = build_ok;
1772   Cmd_line rule, cmd_tail, command = NULL;
1773   char mbstring[MAXPATHLEN];
1774   int filed;
1775   Name target = line->body.line.target;

1777   SEND_MTOOL_MSG(
1778     if (!sent_rsrc_info_msg) {
1779       if (userName[0] == '\0') {
1780         avo_get_user(userName, NULL);
1781       }
1782       if (hostName[0] == '\0') {
1783         strcpy(hostName, avo_hostname());
1784       }
1785       send_rsrc_info_msg(1, hostName, userName);
1786       sent_rsrc_info_msg = 1;
1787     }
1788     send_job_start_msg(line);
1789     job_result_msg = new Avo_MToolJobResultMsg();
1790   );

1792   target->has_recursive_dependency = false;
1793   // We have to create a copy of the rules chain for processing because
1794   // the original one can be destroyed during .make.state file rereading.
1795   for (rule = line->body.line.command_used;
1796        rule != NULL;
1797        rule = rule->next) {
1798     if (command == NULL) {
1799       command = cmd_tail = ALLOC(Cmd_line);
1800     } else {
1801       cmd_tail->next = ALLOC(Cmd_line);
1802       cmd_tail = cmd_tail->next;
1803     }
1804     *cmd_tail = *rule;
1805   }
1806   if (command) {
1807     cmd_tail->next = NULL;
1808   }
1809   for (rule = command; rule != NULL; rule = rule->next) {
1810     if (posix && (touch || quest) && !rule->always_exec) {
1811       continue;
1812     }
1813     if (vpath_defined) {
1814       rule->command_line =
1815       vpath_translation(rule->command_line);
1816     }
1817     /* Echo command line, maybe. */

```

```

1818     if ((rule->command_line->hash.length > 0) &&
1819         !silent &&
1820         (!rule->silent || do_not_exec_rule) &&
1821         (report_dependencies_level == 0)) {
1822         (void) printf("%s\n", rule->command_line->string_mb);
1823         SEND_MTOOL_MSG(
1824             job_result_msg->appendOutput(AVO_STRDUP(rule->co
1825         ));
1826     }
1827     if (rule->command_line->hash.length > 0) {
1828         SEND_MTOOL_MSG(
1829             (void) sprintf(mbstring,
1830                 NOCATGETS("%s/make.stdout.%d.%d.
1831                 tmpdir,
1832                 getpid(),
1833                 file_number++);
1834
1835             int tmp_fd = mkstemp(mbstring);
1836             if(tmp_fd) {
1837                 (void) close(tmp_fd);
1838             }
1839
1840             stdout_file = strdup(mbstring);
1841             stderr_file = NULL;
1842             child_pid = pollResults(stdout_file,
1843                 (char *)NULL,
1844                 (char *)NULL);
1845         );
1846         /* Do assignment if command line prefixed with "=" */
1847         if (rule->assign) {
1848             result = build_ok;
1849             do_assign(rule->command_line, target);
1850         } else if (report_dependencies_level == 0) {
1851             /* Execute command line. */
1852             setvar_envvar();
1853             result = dosys(rule->command_line,
1854                 (Boolean) rule->ignore_error,
1855                 (Boolean) rule->make_refd,
1856                 /* ds 98.04.23 bug #4085164. make
1857                 false,
1858                 /* BOOLEAN(rule->silent &&
1859                 rule->ignore_error), */
1860                 (Boolean) rule->always_exec,
1861                 target,
1862                 send_mtool_msgs);
1863             check_state(temp_file_name);
1864         }
1865         SEND_MTOOL_MSG(
1866             append_job_result_msg(job_result_msg);
1867             if (child_pid > 0) {
1868                 kill(child_pid, SIGUSR1);
1869                 while (!(waitpid(child_pid, 0, 0) == -1
1870                     && (errno == ECHILD)));
1871             }
1872             child_pid = 0;
1873             (void) unlink(stdout_file);
1874             retmem_mb(stdout_file);
1875             stdout_file = NULL;
1876         );
1877     } else {
1878         result = build_ok;
1879     }
1880     if (result == build_failed) {
1881         if (silent || rule->silent) {
1882             (void) printf(catgets(catd, 1, 242, "The followi
1883             rule->command_line->string_mb);

```

```

1884         SEND_MTOOL_MSG(
1885             job_result_msg->appendOutput(AVO_STRDUP(
1886             job_result_msg->appendOutput(AVO_STRDUP(
1887         ));
1888     }
1889     if (!rule->ignore_error && !ignore_errors) {
1890         if (!continue_after_error) {
1891             SEND_MTOOL_MSG(
1892                 job_result_msg->setResult(job_ms
1893                 xdr_msg = (RWCollectable*)
1894                 job_result_msg;
1895                 xdr(&xdrs, xdr_msg);
1896                 (void) fflush(mtool_msgs_fp);
1897                 delete job_result_msg;
1898             );
1899             fatal(catgets(catd, 1, 244, "Command fai
1900             target->string_mb);
1901         }
1902         /*
1903         * Make sure a failing command is not
1904         * saved in .make.state.
1905         */
1906         line->body.line.command_used = NULL;
1907         break;
1908     } else {
1909         result = build_ok;
1910     }
1911 }
1912 }
1913 for (rule = command; rule != NULL; rule = cmd_tail) {
1914     cmd_tail = rule->next;
1915     free(rule);
1916 }
1917 command = NULL;
1918 SEND_MTOOL_MSG(
1919     job_result_msg->setResult(job_msg_id, (result == build_ok) ? 0 :
1920     xdr_msg = (RWCollectable*) job_result_msg;
1921     xdr(&xdrs, xdr_msg);
1922     (void) fflush(mtool_msgs_fp);
1923
1924     delete job_result_msg;
1925 );
1926 if (temp_file_name != NULL) {
1927     free_name(temp_file_name);
1928 }
1929 temp_file_name = NULL;
1930
1931 Property spro = get_prop(sunpro_dependencies->prop, macro_prop);
1932 if (spro != NULL) {
1933     Name val = spro->body.macro.value;
1934     if (val != NULL) {
1935         free_name(val);
1936         spro->body.macro.value = NULL;
1937     }
1938 }
1939 spro = get_prop(sunpro_dependencies->prop, env_mem_prop);
1940 if (spro) {
1941     char *val = spro->body.env_mem.value;
1942     if (val != NULL) {
1943         /*
1944         * Do not return memory allocated for SUNPRO_DEPENDENCIE
1945         * It will be returned in setvar_daemon() in macro.cc
1946         */
1947         // retmem_mb(val);
1948         spro->body.env_mem.value = NULL;
1949     }

```



```

1950     }
1951
1952     return result;
1953 }

1957 /*
1958 *     vpath_translation(cmd)
1959 *
1960 *     Translates one command line by
1961 *     checking each word. If the word has an alias it is translated.
1962 *
1963 *     Return value:
1964 *
1965 *         The translated command
1966 *
1967 *     Parameters:
1968 *         cmd           Command to translate
1969 *
1970 *     Global variables used:
1971 */
1972 Name
1973 vpath_translation(register Name cmd)
1974 {
1975     wchar_t      buffer[STRING_BUFFER_LENGTH];
1976     String_rec   new_cmd;
1977     wchar_t      *p;
1978     wchar_t      *start;

1979     if (!vpath_defined || (cmd == NULL) || (cmd->hash.length == 0)) {
1980         return cmd;
1981     }
1982     INIT_STRING_FROM_STACK(new_cmd, buffer);

1984     Wstring wcb(cmd);
1985     p = wcb.get_string();

1987     while (*p != (int) nul_char) {
1988         while (iswspace(*p) && (*p != (int) nul_char)) {
1989             append_char(*p++, &new_cmd);
1990         }
1991         start = p;
1992         while (!iswspace(*p) && (*p != (int) nul_char)) {
1993             p++;
1994         }
1995         cmd = GETNAME(start, p - start);
1996         if (cmd->has_vpath_alias_prop) {
1997             cmd = get_prop(cmd->prop, vpath_alias_prop)->
1998                 body.vpath_alias.alias;
1999             APPEND_NAME(cmd,
2000                 &new_cmd,
2001                 (int) cmd->hash.length);
2002         } else {
2003             append_string(start, &new_cmd, p - start);
2004         }
2005     }
2006     cmd = GETNAME(new_cmd.buffer.start, FIND_LENGTH);
2007     if (new_cmd.free_after_use) {
2008         retmem(new_cmd.buffer.start);
2009     }
2010     return cmd;
2011 }

2013 /*
2014 *     check_state(temp_file_name)
2015 *

```

```

2016 *     Reads and checks the state changed by the previously executed command.
2017 *
2018 *     Parameters:
2019 *         temp_file_name  The auto dependency temp file
2020 *
2021 *     Global variables used:
2022 */
2023 void
2024 check_state(Name temp_file_name)
2025 {
2026     if (!keep_state) {
2027         return;
2028     }

2030     /*
2031     * Then read the temp file that now might
2032     * contain dependency reports from utilities
2033     */
2034     read_dependency_file(temp_file_name);

2036     /*
2037     * And reread .make.state if it
2038     * changed (the command ran recursive makes)
2039     */
2040     check_read_state_file();
2041     if (temp_file_name != NULL) {
2042         (void) unlink(temp_file_name->string_mb);
2043     }
2044 }

2046 /*
2047 *     read_dependency_file(filename)
2048 *
2049 *     Read the temp file used for reporting dependencies to make
2050 *
2051 *     Parameters:
2052 *         filename        The name of the file with the state info
2053 *
2054 *     Global variables used:
2055 *         makefile_type   The type of makefile being read
2056 *         read_trace_level Debug flag
2057 *         temp_file_number The always increasing number for unique files
2058 *         trace_reader    Debug flag
2059 */
2060 static void
2061 read_dependency_file(register Name filename)
2062 {
2063     register Makefile_type save_makefile_type;

2065     if (filename == NULL) {
2066         return;
2067     }
2068     filename->stat.time = file_no_time;
2069     if (exists(filename) > file_doesnt_exist) {
2070         save_makefile_type = makefile_type;
2071         makefile_type = reading_cpp_file;
2072         if (read_trace_level > 1) {
2073             trace_reader = true;
2074         }
2075         temp_file_number++;
2076         (void) read_simple_file(filename,
2077             false,
2078             false,
2079             false,
2080             false,
2081             false,

```

```

2082         false);
2083     trace_reader = false;
2084     makefile_type = save_makefile_type;
2085 }
2086 }

2088 /*
2089 *   check_read_state_file()
2090 *
2091 *   Check if .make.state has changed
2092 *   If it has we reread it
2093 *
2094 *   Parameters:
2095 *
2096 *   Global variables used:
2097 *       make_state      Make state file name
2098 *       makefile_type   Type of makefile being read
2099 *       read_trace_level Debug flag
2100 *       trace_reader    Debug flag
2101 */
2102 static void
2103 check_read_state_file(void)
2104 {
2105     timestruc_t      previous = make_state->stat.time;
2106     register Makefile_type save_makefile_type;
2107     register Property makefile;

2109     make_state->stat.time = file_no_time;
2110     if ((exists(make_state) == file_doesnt_exist) ||
2111         (make_state->stat.time == previous)) {
2112         return;
2113     }
2114     save_makefile_type = makefile_type;
2115     makefile_type = rereading_statefile;
2116     /* Make sure we clear the old cached contents of .make.state */
2117     makefile = maybe_append_prop(make_state, makefile_prop);
2118     if (makefile->body.makefile.contents != NULL) {
2119         retmem(makefile->body.makefile.contents);
2120         makefile->body.makefile.contents = NULL;
2121     }
2122     if (read_trace_level > 1) {
2123         trace_reader = true;
2124     }
2125     temp_file_number++;
2126     (void) read_simple_file(make_state,
2127         false,
2128         false,
2129         false,
2130         false,
2131         false,
2132         true);
2133     trace_reader = false;
2134     makefile_type = save_makefile_type;
2135 }

2137 /*
2138 *   do_assign(line, target)
2139 *
2140 *   Handles runtime assignments for command lines prefixed with "=".
2141 *
2142 *   Parameters:
2143 *       line      The command that contains an assignment
2144 *       target    The Name of the target, used for error reports
2145 *
2146 *   Global variables used:
2147 *       assign_done Set to indicate doname needs to reprocess

```

```

2148 */
2149 static void
2150 do_assign(register Name line, register Name target)
2151 {
2152     Wstring wcb(line);
2153     register wchar_t *string = wcb.get_string();
2154     register wchar_t *equal;
2155     register Name name;
2156     register Boolean append = false;

2158     /*
2159     * If any runtime assignments are done, doname() must reprocess all
2160     * targets in the future since the macro values used to build the
2161     * command lines for the targets might have changed.
2162     */
2163     assign_done = true;
2164     /* Skip white space. */
2165     while (iswspace(*string)) {
2166         string++;
2167     }
2168     equal = string;
2169     /* Find "+=" or "=". */
2170     while (!iswspace(*equal) &&
2171         (*equal != (int) plus_char) &&
2172         (*equal != (int) equal_char)) {
2173         equal++;
2174     }
2175     /* Internalize macro name. */
2176     name = GETNAME(string, equal - string);
2177     /* Skip over "+=" or "=". */
2178     while (!((*equal == (int) nul_char) ||
2179         (*equal == (int) equal_char) ||
2180         (*equal == (int) plus_char))) {
2181         equal++;
2182     }
2183     switch (*equal) {
2184     case nul_char:
2185         fatal(catgets(catd, 1, 31, "= expected in rule '%s' for target `
2186             line->string_mb,
2187             target->string_mb);
2188     case plus_char:
2189         append = true;
2190         equal++;
2191         break;
2192     }
2193     equal++;
2194     /* Skip over whitespace in front of value. */
2195     while (iswspace(*equal)) {
2196         equal++;
2197     }
2198     /* Enter new macro value. */
2199     enter_equal(name,
2200         GETNAME(equal, wcb.get_string() + line->hash.length - equal)
2201         append);
2202 }

2204 /*
2205 *   build_command_strings(target, line)
2206 *
2207 *   Builds the command string to used when
2208 *   building a target. If the string is different from the previous one
2209 *   is_out_of_date is set.
2210 *
2211 *   Parameters:
2212 *       target    Target to build commands for
2213 *       line      Where to stuff result

```

```

2214 *
2215 *   Global variables used:
2216 *       c_at           The Name "@", used to set macro value
2217 *       command_changed Set if command is different from old
2218 *       debug_level    Should we trace activities?
2219 *       do_not_exec_rule Always echo when running -n
2220 *       empty_name     The Name "", used for empty rule
2221 *       funny          Semantics of characters
2222 *       ignore_errors  Used to init field for line
2223 *       is_conditional Set to false before evaling macro, checked
2224 *                   after expanding macros
2225 *       keep_state     Indicates that .KEEP_STATE is on
2226 *       make_word_mentioned Set by macro eval, inits field for cmd
2227 *       query          The Name "?", used to set macro value
2228 *       query_mentioned Set by macro eval, inits field for cmd
2229 *       recursion_level Used for tracing
2230 *       silent         Used to init field for line
2231 */
2232 static void
2233 build_command_strings(Name target, register Property line)
2234 {
2235     String_rec      command_line;
2236     register Cmd_line command_template = line->body.line.command_tmpl
2237     register Cmd_line *insert = &line->body.line.command_used;
2238     register Cmd_line used = *insert;
2239     wchar_t         buffer[STRING_BUFFER_LENGTH];
2240     wchar_t         *start;
2241     Name            new_command_line;
2242     register Boolean new_command_longer = false;
2243     register Boolean ignore_all_command_dependency = true;
2244     Property        member;
2245     static Name     less_name;
2246     static Name     percent_name;
2247     static Name     star;
2248     Name            tmp_name;

2250     if (less_name == NULL) {
2251         MBSTOWCS(wcs_buffer, "<");
2252         less_name = GETNAME(wcs_buffer, FIND_LENGTH);
2253         MBSTOWCS(wcs_buffer, "%");
2254         percent_name = GETNAME(wcs_buffer, FIND_LENGTH);
2255         MBSTOWCS(wcs_buffer, "*");
2256         star = GETNAME(wcs_buffer, FIND_LENGTH);
2257     }

2259     /* We have to check if a target depends on conditional macros */
2260     /* Targets that do must be reprocessed by doname() each time around */
2261     /* since the macro values used when building the target might have */
2262     /* changed */
2263     conditional_macro_used = false;
2264     /* If we are building a lib.a(member) target $@ should be bound */
2265     /* to lib.a */
2266     if (target->is_member &&
2267         ((member = get_prop(target->prop, member_prop)) != NULL)) {
2268         target = member->body.member.library;
2269     }
2270     /* If we are building a "::" help target $@ should be bound to */
2271     /* the real target name */
2272     /* A lib.a(member) target is never :: */
2273     if (target->has_target_prop) {
2274         target = get_prop(target->prop, target_prop)->
2275             body.target.target;
2276     }
2277     /* Bind the magic macros that make supplies */
2278     tmp_name = target;
2279     if(tmp_name != NULL) {

```

```

2280         if (tmp_name->has_vpath_alias_prop) {
2281             tmp_name = get_prop(tmp_name->prop, vpath_alias_prop)->
2282                 body.vpath_alias.alias;
2283         }
2284     }
2285     (void) SETVAR(c_at, tmp_name, false);

2287     tmp_name = line->body.line.star;
2288     if(tmp_name != NULL) {
2289         if (tmp_name->has_vpath_alias_prop) {
2290             tmp_name = get_prop(tmp_name->prop, vpath_alias_prop)->
2291                 body.vpath_alias.alias;
2292         }
2293     }
2294     (void) SETVAR(star, tmp_name, false);

2296     tmp_name = line->body.line.less;
2297     if(tmp_name != NULL) {
2298         if (tmp_name->has_vpath_alias_prop) {
2299             tmp_name = get_prop(tmp_name->prop, vpath_alias_prop)->
2300                 body.vpath_alias.alias;
2301         }
2302     }
2303     (void) SETVAR(less_name, tmp_name, false);

2305     tmp_name = line->body.line.percent;
2306     if(tmp_name != NULL) {
2307         if (tmp_name->has_vpath_alias_prop) {
2308             tmp_name = get_prop(tmp_name->prop, vpath_alias_prop)->
2309                 body.vpath_alias.alias;
2310         }
2311     }
2312     (void) SETVAR(percent_name, tmp_name, false);

2314     /* $? is seldom used and it is expensive to build */
2315     /* so we store the list form and build the string on demand */
2316     Chain query_list = NULL;
2317     Chain *query_list_tail = &query_list;

2319     for (Chain ch = line->body.line.query; ch != NULL; ch = ch->next) {
2320         *query_list_tail = ALLOC(Chain);
2321         (*query_list_tail)->name = ch->name;
2322         if ((*query_list_tail)->name->has_vpath_alias_prop) {
2323             (*query_list_tail)->name =
2324                 get_prop((*query_list_tail)->name->prop,
2325                     vpath_alias_prop)->body.vpath_alias.alias;
2326         }
2327         (*query_list_tail)->next = NULL;
2328         query_list_tail = &(*query_list_tail)->next;
2329     }
2330     (void) setvar_daemon(query,
2331         (Name) query_list,
2332         false,
2333         chain_daemon,
2334         false,
2335         debug_level);

2337     /* build $^ */
2338     Chain hat_list = NULL;
2339     Chain *hat_list_tail = &hat_list;

2341     for (Dependency dependency = line->body.line.dependencies;
2342          dependency != NULL;
2343          dependency = dependency->next) {
2344         /* skip automatic dependencies */
2345         if (!dependency->automatic) {

```

```

2346         if ((dependency->name != force) &&
2347             (dependency->stale == false)) {
2348             *hat_list_tail = ALLOC(Chain);
2349
2350             if (dependency->name->is_member &&
2351                 (get_prop(dependency->name->prop, member
2352                     (*hat_list_tail)->name =
2353                         get_prop(dependency->name
2354                             member_prop)->bo
2355                     } else {
2356                         (*hat_list_tail)->name = dependency->name
2357                     }
2358
2359             if ((*hat_list_tail)->name != NULL) {
2360                 if ((*hat_list_tail)->name->has_vpath_al
2361                     (*hat_list_tail)->name =
2362                         get_prop((*hat_list_tail
2363                             vpath_alias_prop
2364                     }
2365             }
2366
2367             (*hat_list_tail)->next = NULL;
2368             hat_list_tail = &(*hat_list_tail)->next;
2369         }
2370     }
2371 }
2372 (void) setvar_daemon(hat,
2373     (Name) hat_list,
2374     false,
2375     chain_daemon,
2376     false,
2377     debug_level);
2378
2379 /* We have two command sequences we need to handle */
2380 /* The old one that we probably read from .make.state */
2381 /* and the new one we are building that will replace the old one */
2382 /* Even when KEEP_STATE is not on we build a new command sequence and store */
2383 /* it in the line prop. This command sequence is then executed by */
2384 /* run_command(). If KEEP_STATE is on it is also later written to */
2385 /* .make.state. The routine replaces the old command line by line with the */
2386 /* new one trying to reuse Cmd_lines */
2387
2388 /* If there is no old command_used we have to start creating */
2389 /* Cmd_lines to keep the new cmd in */
2390 if (used == NULL) {
2391     new_command_longer = true;
2392     *insert = used = ALLOC(Cmd_line);
2393     used->next = NULL;
2394     used->command_line = NULL;
2395     insert = &used->next;
2396 }
2397 /* Run thru the template for the new command and build the expanded */
2398 /* new command lines */
2399 for (;
2400     command_template != NULL;
2401     command_template = command_template->next, insert = &used->next, us
2402     /* If there is no old command_used Cmd_line we need to */
2403     /* create one and say that cmd consistency failed */
2404     if (used == NULL) {
2405         new_command_longer = true;
2406         *insert = used = ALLOC(Cmd_line);
2407         used->next = NULL;
2408         used->command_line = empty_name;
2409     }
2410     /* Prepare the Cmd_line for the processing */
2411     /* The command line prefixes "@-?" are stripped and that */

```

```

2412     /* information is saved in the Cmd_line */
2413     used->assign = false;
2414     used->ignore_error = ignore_errors;
2415     used->silent = silent;
2416     used->always_exec = false;
2417     /* Expand the macros in the command line */
2418     INIT_STRING_FROM_STACK(command_line, buffer);
2419     make_word_mentioned =
2420     query_mentioned =
2421     false;
2422     expand_value(command_template->command_line, &command_line, true
2423     /* If the macro $(MAKE) is mentioned in the command */
2424     /* "make -n" runs actually execute the command */
2425     used->make_refd = make_word_mentioned;
2426     used->ignore_command_dependency = query_mentioned;
2427     /* Strip the prefixes */
2428     start = command_line.buffer.start;
2429     for (;
2430         iswspace(*start) ||
2431         (get_char_semantics_value(*start) & (int) command_prefix_se
2432         start++) {
2433         switch (*start) {
2434             case question_char:
2435                 used->ignore_command_dependency = true;
2436                 break;
2437             case exclam_char:
2438                 used->ignore_command_dependency = false;
2439                 break;
2440             case equal_char:
2441                 used->assign = true;
2442                 break;
2443             case hyphen_char:
2444                 used->ignore_error = true;
2445                 break;
2446             case at_char:
2447                 if (!do_not_exec_rule) {
2448                     used->silent = true;
2449                 }
2450                 break;
2451             case plus_char:
2452                 if (posix) {
2453                     used->always_exec = true;
2454                 }
2455                 break;
2456             }
2457         }
2458     /* If all command lines of the template are prefixed with "*/
2459     /* the VIRTUAL_ROOT is not used for cmd consistency checks */
2460     if (!used->ignore_command_dependency) {
2461         ignore_all_command_dependency = false;
2462     }
2463     /* Internalize the expanded and stripped command line */
2464     new_command_line = GETNAME(start, FIND_LENGTH);
2465     if ((used->command_line == NULL) &&
2466         (line->body.line.sccc_command)) {
2467         used->command_line = new_command_line;
2468         new_command_longer = false;
2469     }
2470     /* Compare it with the old one for command consistency */
2471     if (used->command_line != new_command_line) {
2472         Name vpath_translated = vpath_translation(new_command_li
2473         if (keep_state &&
2474             !used->ignore_command_dependency && (vpath_translate
2475             if (debug_level > 0) {
2476                 if (used->command_line != NULL
2477                     && *used->command_line->string_mb !=

```

```

2478         '\0' {
2479             (void) printf(catgets(catd, 1, 3
2480                 recursion_level,
2481                 "",
2482                 target->string_mb,
2483                 vpath_translated->
2484                 recursion_level,
2485                 "",
2486                 used->
2487                 command_line->
2488                 string_mb);
2489         } else {
2490             (void) printf(catgets(catd, 1, 3
2491                 recursion_level,
2492                 "",
2493                 target->string_mb,
2494                 vpath_translated->
2495                 recursion_level,
2496                 "");
2497         }
2498     }
2499     command_changed = true;
2500     line->body.line.is_out_of_date = true;
2501 }
2502 used->command_line = new_command_line;
2503 }
2504 if (command_line.free_after_use) {
2505     retmem(command_line.buffer.start);
2506 }
2507 }
2508 /* Check if the old command is longer than the new for */
2509 /* command consistency */
2510 if (used != NULL) {
2511     *insert = NULL;
2512     if (keep_state &&
2513         !ignore_all_command_dependency) {
2514         if (debug_level > 0) {
2515             (void) printf(catgets(catd, 1, 34, "%sBuilding
2516                 recursion_level,
2517                 "",
2518                 target->string_mb);
2519         }
2520         command_changed = true;
2521         line->body.line.is_out_of_date = true;
2522     }
2523 }
2524 /* Check if the new command is longer than the old command for */
2525 /* command consistency */
2526 if (new_command_longer &&
2527     !ignore_all_command_dependency &&
2528     keep_state) {
2529     if (debug_level > 0) {
2530         (void) printf(catgets(catd, 1, 35, "%sBuilding %s becau
2531             recursion_level,
2532             "",
2533             target->string_mb);
2534     }
2535     command_changed = true;
2536     line->body.line.is_out_of_date = true;
2537 }
2538 /* Unbind the magic macros */
2539 (void) SETVAR(c_at, (Name) NULL, false);
2540 (void) SETVAR(star, (Name) NULL, false);
2541 (void) SETVAR(less_name, (Name) NULL, false);
2542 (void) SETVAR(percent_name, (Name) NULL, false);
2543 (void) SETVAR(query, (Name) NULL, false);

```

```

2544     if (query_list != NULL) {
2545         delete_query_chain(query_list);
2546     }
2547     (void) SETVAR(hat, (Name) NULL, false);
2548     if (hat_list != NULL) {
2549         delete_query_chain(hat_list);
2550     }
2551 }
2552 if (conditional_macro_used) {
2553     target->conditional_macro_list = cond_macro_list;
2554     cond_macro_list = NULL;
2555     target->depends_on_conditional = true;
2556 }
2557 }
2558 }
2559 /*
2560 * touch_command(line, target, result)
2561 *
2562 * If this is an "make -t" run we do this.
2563 * We touch all targets in the target group ("foo + fie:") if any.
2564 *
2565 * Return value:
2566 * Indicates if the command failed or not
2567 *
2568 * Parameters:
2569 * line The command line to update
2570 * target The target we are touching
2571 * result Initial value for the result we return
2572 *
2573 * Global variables used:
2574 * do_not_exec_rule Indicates that -n is on
2575 * silent Do not echo commands
2576 */
2577 static Doname
2578 touch_command(register Property line, register Name target, Doname result)
2579 {
2580     Name name;
2581     register Chain target_group;
2582     String_rec touch_string;
2583     wchar_t buffer[MAXPATHLEN];
2584     Name touch_cmd;
2585     Cmd_line rule;
2586 }
2587
2588 SEND_MTOOL_MSG(
2589     if (!sent_rsrc_info_msg) {
2590         if (userName[0] == '\0') {
2591             avo_get_user(userName, NULL);
2592         }
2593         if (hostName[0] == '\0') {
2594             strcpy(hostName, avo_hostname());
2595         }
2596         send_rsrc_info_msg(1, hostName, userName);
2597         sent_rsrc_info_msg = 1;
2598     }
2599     send_job_start_msg(line);
2600     job_result_msg = new Avo_MToolJobResultMsg();
2601 );
2602 for (name = target, target_group = NULL; name != NULL;) {
2603     if (!name->is_member) {
2604         /*
2605          * Build a touch command that can be passed
2606          * to dosys(). If KEEP_STATE is on, "make -t"
2607          * will save the proper command, not the
2608          * "touch" in .make.state.
2609          */

```

```

2610 INIT_STRING_FROM_STACK(touch_string, buffer);
2611 MBSTOWCS(wcs_buffer, NOCATGETS("touch "));
2612 append_string(wcs_buffer, &touch_string, FIND_LENGTH);
2613 touch_cmd = name;
2614 if (name->has_vpath_alias_prop) {
2615     touch_cmd = get_prop(name->prop,
2616         vpath_alias_prop->
2617         body.vpath_alias.alias;
2618 }
2619 APPEND_NAME(touch_cmd,
2620     &touch_string,
2621     FIND_LENGTH);
2622 touch_cmd = GETNAME(touch_string.buffer.start,
2623     FIND_LENGTH);
2624 if (touch_string.free_after_use) {
2625     retmem(touch_string.buffer.start);
2626 }
2627 if (!silent ||
2628     do_not_exec_rule &&
2629     (target_group == NULL)) {
2630     (void) printf("%s\n", touch_cmd->string_mb);
2631     SEND_MTOOL_MSG(
2632         job_result_msg->appendOutput(AVO_STRDUP(
2633         ));
2634 }
2635 /* Run the touch command, or simulate it */
2636 if (!do_not_exec_rule) {
2637
2638     SEND_MTOOL_MSG(
2639         (void) sprintf(mbstring,
2640             NOCATGETS("%s/make.stdou
2641             tmpdir,
2642             getpid(),
2643             file_number++);
2644
2645         int tmp_fd = mkstemp(mbstring);
2646         if(tmp_fd) {
2647             (void) close(tmp_fd);
2648         }
2649
2650         stdout_file = strdup(mbstring);
2651         stderr_file = NULL;
2652         child_pid = pollResults(stdout_file,
2653             (char *)NULL,
2654             (char *)NULL);
2655     );
2656
2657     result = dosys(touch_cmd,
2658         false,
2659         false,
2660         false,
2661         false,
2662         name,
2663         send_mtool_msgs);
2664
2665     SEND_MTOOL_MSG(
2666         append_job_result_msg(job_result_msg);
2667         if (child_pid > 0) {
2668             kill(child_pid, SIGUSR1);
2669             while (!(waitpid(child_pid, 0,
2670                 && (errno == ECHILD))));
2671         }
2672         child_pid = 0;
2673         (void) unlink(stdout_file);
2674         retmem_mb(stdout_file);
2675         stdout_file = NULL;

```

```

2676     );
2677
2678     } else {
2679         result = build_ok;
2680     }
2681 } else {
2682     result = build_ok;
2683 }
2684 if (target_group == NULL) {
2685     target_group = line->body.line.target_group;
2686 } else {
2687     target_group = target_group->next;
2688 }
2689 if (target_group != NULL) {
2690     name = target_group->name;
2691 } else {
2692     name = NULL;
2693 }
2694 }
2695 SEND_MTOOL_MSG(
2696     job_result_msg->setResult(job_msg_id, (result == build_ok) ? 0 :
2697     xdr_msg = (RWCollectable*) job_result_msg;
2698     xdr(&xdrs, xdr_msg);
2699     (void) fflush(mtool_msgs_fp);
2700     delete job_result_msg;
2701 );
2702 return result;
2703 }
2704
2705 /*
2706 * update_target(line, result)
2707 *
2708 * updates the status of a target after executing its commands.
2709 *
2710 * Parameters:
2711 *     line           The command line block to update
2712 *     result         Indicates that build is OK so can update
2713 *
2714 * Global variables used:
2715 *     do_not_exec_rule Indicates that -n is on
2716 *     touch             Fake the new timestamp if we are just touching
2717 */
2718 void
2719 update_target(Property line, Doname result)
2720 {
2721     Name target;
2722     Chain target_group;
2723     Property line2;
2724     timestruc_t old_stat_time;
2725     Property member;
2726
2727     /*
2728     * [tolik] Additional fix for bug 1063790. It was fixed
2729     * for serial make long ago, but DMake dumps core when
2730     * target is a symlink and sccs file is newer then target.
2731     * In this case, finish_children() calls update_target()
2732     * with line==NULL.
2733     */
2734     if(line == NULL) {
2735         /* XXX. Should we do anything here? */
2736         return;
2737     }
2738
2739     target = line->body.line.target;
2740
2741     if ((result == build_ok) && (line->body.line.command_used != NULL)) {

```

```

2742     if (do_not_exec_rule ||
2743         touch ||
2744         (target->is_member &&
2745          (line->body.line.command_template != NULL) &&
2746          (line->body.line.command_template->command_line->string_mb[
2747           (line->body.line.command_template->next == NULL))) {
2748         /* If we are simulating execution we need to fake a */
2749         /* new timestamp for the target we didnt build */
2750         target->stat.time = file_max_time;
2751     } else {
2752         /*
2753          * If we really built the target we read the new
2754          * timestamp.
2755          * Fix for bug #1110906: if .c file is newer than
2756          * the corresponding .o file which is in an archive
2757          * file, make will compile the .c file but it won't
2758          * update the object in the .a file.
2759          */
2760         old_stat_time = target->stat.time;
2761         target->stat.time = file_no_time;
2762         (void) exists(target);
2763         if ((target->is_member) &&
2764             (target->stat.time == old_stat_time)) {
2765             member = get_prop(target->prop, member_prop);
2766             if (member != NULL) {
2767                 target->stat.time = member->body.member.
2768                 target->stat.time.tv_sec++;
2769             }
2770         }
2771     }
2772     /* If the target is part of a group we need to propagate the */
2773     /* result of the run to all members */
2774     for (target_group = line->body.line.target_group;
2775          target_group != NULL;
2776          target_group = target_group->next) {
2777         target_group->name->stat.time = target->stat.time;
2778         line2 = maybe_append_prop(target_group->name,
2779                                 line_prop);
2780         line2->body.line.command_used =
2781         line->body.line.command_used;
2782         line2->body.line.target = target_group->name;
2783     }
2784     target->has_built = true;
2785 }
2786 }

2788 /*
2789 * sccs_get(target, command)
2790 *
2791 * Figures out if it possible to sccs get a file
2792 * and builds the command to do it if it is.
2793 *
2794 * Return value:
2795 *             Indicates if sccs get failed or not
2796 *
2797 * Parameters:
2798 *     target   Target to get
2799 *     command  Where to deposit command to use
2800 *
2801 * Global variables used:
2802 *     debug_level  Should we trace activities?
2803 *     recursion_level Used for tracing
2804 *     sccs_get_rule The rule to used for sccs getting
2805 */
2806 static Doname
2807 sccs_get(register Name target, register Property *command)

```

```

2808 {
2809     register int     result;
2810     char             link[MAXPATHLEN];
2811     String_rec       string;
2812     wchar_t          name[MAXPATHLEN];
2813     register wchar_t *p;
2814     timestruc_t      sccs_time;
2815     register Property line;
2816     int              sym_link_depth = 0;

2818     /* For sccs, we need to chase symlinks. */
2819     while (target->stat.is_sym_link) {
2820         if (sym_link_depth++ > 90) {
2821             fatal(catgets(catd, 1, 95, "Can't read symbolic link '%s
2822                 target->string_mb);
2823         }
2824         /* Read the value of the link. */
2825         result = readlink_vroot(target->string_mb,
2826                                link,
2827                                sizeof(link),
2828                                NULL,
2829                                VROOT_DEFAULT);
2830         if (result == -1) {
2831             fatal(catgets(catd, 1, 36, "Can't read symbolic link '%s
2832                 target->string_mb, errmsg(errno));
2833         }
2834         link[result] = 0;
2835         /* Use the value to build the proper filename. */
2836         INIT_STRING_FROM_STACK(string, name);

2838         Wstring wcb(target);
2839         if ((link[0] != slash_char) &&
2840             ((p = (wchar_t *) wsrchr(wcb.get_string(), slash_char)) != N
2841              append_string(wcb.get_string(), &string, p - wcb.get_str
2842         )
2843         append_string(link, &string, result);
2844         /* Replace the old name with the translated name. */
2845         target = normalize_name(string.buffer.start, string.text.p - str
2846         (void) exists(target);
2847         if (string.free_after_use) {
2848             retmem(string.buffer.start);
2849         }
2850     }

2852     /*
2853     * read_dir() also reads the ?/SCCS dir and saves information
2854     * about which files have SCSC/s. files.
2855     */
2856     if (target->stat.has_sccs == DONT_KNOW_SCCS) {
2857         read_directory_of_file(target);
2858     }
2859     switch (target->stat.has_sccs) {
2860     case DONT_KNOW_SCCS:
2861         /* We dont know by now there is no SCCS/s.* */
2862         target->stat.has_sccs = NO_SCCS;
2863     case NO_SCCS:
2864         /*
2865          * If there is no SCCS/s.* but the plain file exists,
2866          * we say things are OK.
2867          */
2868         if (target->stat.time > file_doesnt_exist) {
2869             return build_ok;
2870         }
2871         /* If we cant find the plain file, we give up. */
2872         return build_dont_know;
2873     case HAS_SCCS:

```

```

2874      /*
2875      * Pay dirt. We now need to figure out if the plain file
2876      * is out of date relative to the SCCS/s.* file.
2877      */
2878      sccs_time = exists(get_prop(target->prop,
2879                          sccs_prop->body.sccs.file);
2880      break;
2881  }

2883  if ((!target->has_complained &&
2884      (sccs_time != file_doesnt_exist) &&
2885      (sccs_get_rule != NULL))) {
2886      /* only checking */
2887      if (command == NULL) {
2888          return build_ok;
2889      }
2890      /*
2891      * We provide a command line for the target. The line is a
2892      * "sccs get" command from default.mk.
2893      */
2894      line = maybe_append_prop(target, line_prop);
2895      *command = line;
2896      if (sccs_time > target->stat.time) {
2897          /*
2898          * And only if the plain file is out of date do we
2899          * request execution of the command.
2900          */
2901          line->body.line.is_out_of_date = true;
2902          if (debug_level > 0) {
2903              (void) printf(catgets(catd, 1, 37, "%sSCCs gett
2904                          recursion_level,
2905                          "",
2906                          target->string_mb);
2907          }
2908      }
2909      line->body.line.sccs_command = true;
2910      line->body.line.command_template = sccs_get_rule;
2911      if (!svr4 && (!allrules_read || posix)) {
2912          if ((target->prop) &&
2913              (target->prop->body.sccs.file) &&
2914              (target->prop->body.sccs.file->string_mb)) {
2915              if ((strlen(target->prop->body.sccs.file->string_mb) ==
2916                  strlen(target->string_mb) + 2) &&
2917                  (target->prop->body.sccs.file->string_mb[0] == 's') &&
2918                  (target->prop->body.sccs.file->string_mb[1] == '.')) {
2919                  line->body.line.command_template = get_posix_rule;
2920              }
2921          }
2922      }
2923      line->body.line.target = target;
2924      /*
2925      * Also make sure the rule is build with $* and $<
2926      * bound properly.
2927      */
2928      line->body.line.star = NULL;
2929      line->body.line.less = NULL;
2930      line->body.line.percent = NULL;
2931      return build_ok;
2932  }
2933  return build_dont_know;
2934  }
2935  }

2937  /*
2938  * read_directory_of_file(file)
2939  *

```

```

2940  * Reads the directory the specified file lives in.
2941  *
2942  * Parameters:
2943  *     file           The file we need to read dir for
2944  *
2945  * Global variables used:
2946  *     dot           The Name ".", used as the default dir
2947  */
2948  void
2949  read_directory_of_file(register Name file)
2950  {
2951
2952      Wstring file_string(file);
2953      wchar_t * wcb = file_string.get_string();
2954      wchar_t usr_include_buf[MAXPATHLEN];
2955      wchar_t usr_include_sys_buf[MAXPATHLEN];
2956
2957      register Name      directory = dot;
2958      register wchar_t   *p = (wchar_t *) wsrchr(wcb,
2959                                                  (int) slash_char);
2960
2961      register int      length = p - wcb;
2962      static Name       usr_include;
2963      static Name       usr_include_sys;
2964
2965      if (usr_include == NULL) {
2966          MBSTOWCS(usr_include_buf, NOCATGETS("/usr/include"));
2967          usr_include = GETNAME(usr_include_buf, FIND_LENGTH);
2968          MBSTOWCS(usr_include_sys_buf, NOCATGETS("/usr/include/sys"));
2969          usr_include_sys = GETNAME(usr_include_sys_buf, FIND_LENGTH);
2970      }
2971
2972      /*
2973      * If the filename contains a "/" we have to extract the path
2974      * Else the path defaults to ".".
2975      */
2976      if (p != NULL) {
2977          /*
2978          * Check some popular directories first to possibly
2979          * save time. Compare string length first to gain speed.
2980          */
2981          if ((usr_include->hash.length == length) &&
2982              IS_WEQUALN(usr_include_buf,
2983                          wcb,
2984                          length)) {
2985              directory = usr_include;
2986          } else if ((usr_include_sys->hash.length == length) &&
2987                      IS_WEQUALN(usr_include_sys_buf,
2988                                  wcb,
2989                                  length)) {
2990              directory = usr_include_sys;
2991          } else {
2992              directory = GETNAME(wcb, length);
2993          }
2994      }
2995      (void) read_dir(directory,
2996                      (wchar_t *) NULL,
2997                      (Property) NULL,
2998                      (wchar_t *) NULL);
2999  }

3000  /*
3001  * add_pattern_conditionals(target)
3002  *
3003  * Scan the list of conditionals defined for pattern targets and add any
3004  * that match this target to its list of conditionals.
3005  *

```



```

3006 *      Parameters:
3007 *          target          The target we should add conditionals for
3008 *
3009 *      Global variables used:
3010 *          conditionals    The list of pattern conditionals
3011 */
3012 static void
3013 add_pattern_conditionals(register Name target)
3014 {
3015     register Property    conditional;
3016     Property            new_prop;
3017     Property            *previous;
3018     Name_rec            dummy;
3019     wchar_t             *pattern;
3020     wchar_t             *percent;
3021     int                 length;
3022
3023     Wstring wcb(target);
3024     Wstring wtbl;
3025
3026     for (conditional = get_prop(conditionals->prop, conditional_prop);
3027          conditional != NULL;
3028          conditional = get_prop(conditional->next, conditional_prop)) {
3029         wtbl.init(conditional->body.conditional.target);
3030         pattern = wtbl.get_string();
3031         if (pattern[1] != 0) {
3032             percent = (wchar_t *) wschr(pattern, (int) percent_char);
3033             if (!wcb.equaln(pattern, percent-pattern) ||
3034                 !IS_WEQUAL(wcb.get_string(wcb.length()-wslen(percent)
3035                                     ),
3036                             continue;
3037         }
3038         for (previous = &target->prop;
3039              *previous != NULL;
3040              previous = &(*previous)->next) {
3041             if (((*previous)->type == conditional_prop) &&
3042                 ((*previous)->body.conditional.sequence >
3043                  conditional->body.conditional.sequence)) {
3044                 break;
3045             }
3046         }
3047         if (*previous == NULL) {
3048             new_prop = append_prop(target, conditional_prop);
3049         } else {
3050             dummy.prop = NULL;
3051             new_prop = append_prop(&dummy, conditional_prop);
3052             new_prop->next = *previous;
3053             *previous = new_prop;
3054         }
3055         target->conditional_cnt++;
3056         new_prop->body.conditional = conditional->body.conditional;
3057     }
3058 }
3059
3060 /*
3061 *      set_locals(target, old_locals)
3062 *
3063 *      Sets any conditional macros for the target.
3064 *      Each target carries a possibly empty set of conditional properties.
3065 *
3066 *      Parameters:
3067 *          target          The target to set conditional macros for
3068 *          old_locals      Space to store old values in
3069 *
3070 *      Global variables used:
3071 *          debug_level     Should we trace activity?

```

```

3072 *          is_conditional We need to preserve this value
3073 *          recursion_level Used for tracing
3074 */
3075 void
3076 set_locals(register Name target, register Property old_locals)
3077 {
3078     register Property    conditional;
3079     register int         i;
3080     register Boolean     saved_conditional_macro_used;
3081     Chain                cond_name;
3082     Chain                cond_chain;
3083
3084     if (target->dont_activate_cond_values) {
3085         return;
3086     }
3087
3088     saved_conditional_macro_used = conditional_macro_used;
3089
3090     /* Scan the list of conditional properties and apply each one */
3091     for (conditional = get_prop(target->prop, conditional_prop), i = 0;
3092          conditional != NULL;
3093          conditional = get_prop(conditional->next, conditional_prop),
3094          i++) {
3095         /* Save the old value */
3096         old_locals[i].body.macro =
3097             maybe_append_prop(conditional->body.conditional.name,
3098                               macro_prop->body.macro;
3099         if (debug_level > 1) {
3100             (void) printf(catgets(catd, 1, 38, "%s*Activating condit
3101                             recursion_level,
3102                             "");
3103         }
3104         /* Set the conditional value. Macros are expanded when the */
3105         /* macro is refd as usual */
3106         if ((conditional->body.conditional.name != virtual_root) ||
3107             (conditional->body.conditional.value != virtual_root)) {
3108             (void) SETVAR(conditional->body.conditional.name,
3109                           conditional->body.conditional.value,
3110                           (Boolean) conditional->body.conditional.ap
3111         }
3112         cond_name = ALLOC(Chain);
3113         cond_name->name = conditional->body.conditional.name;
3114     }
3115     /* Put this target on the front of the chain of conditional targets */
3116     cond_chain = ALLOC(Chain);
3117     cond_chain->name = target;
3118     cond_chain->next = conditional_targets;
3119     conditional_targets = cond_chain;
3120     conditional_macro_used = saved_conditional_macro_used;
3121 }
3122
3123 /*
3124 *      reset_locals(target, old_locals, conditional, index)
3125 *
3126 *      Removes any conditional macros for the target.
3127 *
3128 *      Parameters:
3129 *          target          The target we are restoring values for
3130 *          old_locals      The values to restore
3131 *          conditional      The first conditional block for the target
3132 *          index           into the old_locals vector
3133 *
3134 *      Global variables used:
3135 *          debug_level     Should we trace activities?
3136 *          recursion_level Used for tracing
3137 void

```

```

3138 reset_locals(register Name target, register Property old_locals, register Proper
3139 {
3140     register Property      this_conditional;
3141     Chain                  cond_chain;
3142
3143     if (target->dont_activate_cond_values) {
3144         return;
3145     }
3146
3147     /* Scan the list of conditional properties and restore the old value */
3148     /* to each one Reverse the order relative to when we assigned macros */
3149     this_conditional = get_prop(conditional->next, conditional_prop);
3150     if (this_conditional != NULL) {
3151         reset_locals(target, old_locals, this_conditional, index+1);
3152     } else {
3153         /* Remove conditional target from chain */
3154         if (conditional_targets == NULL ||
3155             conditional_targets->name != target) {
3156             warning(catgets(catd, 1, 39, "Internal error: reset targ
3157         } else {
3158             cond_chain = conditional_targets->next;
3159             retmem_mb((caddr_t) conditional_targets);
3160             conditional_targets = cond_chain;
3161         }
3162     }
3163     get_prop(conditional->body.conditional.name->prop,
3164             macro_prop->body.macro = old_locals[index].body.macro;
3165     if (conditional->body.conditional.name == virtual_root) {
3166         (void) SETVAR(virtual_root, getvar(virtual_root), false);
3167     }
3168     if (debug_level > 1) {
3169         if (old_locals[index].body.macro.value != NULL) {
3170             (void) printf(catgets(catd, 1, 40, "%sdeactivating cond
3171                 recursion_level,
3172                 "",
3173                 conditional->body.conditional.name->
3174                 string_mb,
3175                 old_locals[index].body.macro.value->
3176                 string_mb);
3177         } else {
3178             (void) printf(catgets(catd, 1, 41, "%sdeactivating cond
3179                 recursion_level,
3180                 "",
3181                 conditional->body.conditional.name->
3182                 string_mb);
3183         }
3184     }
3185 }
3186
3187 /*
3188 *   check_auto_dependencies(target, auto_count, automatics)
3189 *
3190 *   Returns true if the target now has a dependency
3191 *   it didn't previously have (saved on automatics).
3192 *
3193 *   Return value:
3194 *       true if new dependency found
3195 *
3196 *   Parameters:
3197 *       target      Target we check
3198 *       auto_count  Number of old automatic vars
3199 *       automatics  Saved old automatics
3200 *
3201 *   Global variables used:
3202 *       keep_state  Indicates that .KEEP_STATE is on
3203 */

```

```

3204 Boolean
3205 check_auto_dependencies(Name target, int auto_count, Name *automatics)
3206 {
3207     Name      *p;
3208     int       n;
3209     Property  line;
3210     Dependency dependency;
3211
3212     if (keep_state) {
3213         if ((line = get_prop(target->prop, line_prop)) == NULL) {
3214             return false;
3215         }
3216         /* Go thru new list of automatic depes */
3217         for (dependency = line->body.line.dependencies;
3218             dependency != NULL;
3219             dependency = dependency->next) {
3220             /* And make sure that each one existed before we */
3221             /* built the target */
3222             if (dependency->automatic && !dependency->stale) {
3223                 for (n = auto_count, p = automatics;
3224                     n > 0;
3225                     n--) {
3226                     if (*p++ == dependency->name) {
3227                         /* If we can find it on the */
3228                         /* saved list of autos we */
3229                         /* are OK */
3230                         goto not_new;
3231                     }
3232                 }
3233                 /* But if we scan over the old list */
3234                 /* of auto. without finding it it is */
3235                 /* new and we must check it */
3236                 return true;
3237             }
3238             not_new;
3239         }
3240         return false;
3241     } else {
3242         return false;
3243     }
3244 }
3245
3246 // Recursively delete each of the Chain struct on the chain.
3247
3248 static void
3249 delete_query_chain(Chain ch)
3250 {
3251     if (ch == NULL) {
3252         return;
3253     } else {
3254         delete_query_chain(ch->next);
3255         retmem_mb((char *) ch);
3256     }
3257 }
3258
3259 Doname
3260 target_can_be_built(register Name target) {
3261     Doname      result = build_dont_know;
3262     Name        true_target = target;
3263     Property    line;
3264
3265     if (target == wait_name) {
3266         return(build_ok);
3267     }
3268 }
3269 /*

```

```

3270     * If the target is a constructed one for a "::" target,
3271     * we need to consider that.
3272     */
3273     if (target->has_target_prop) {
3274         true_target = get_prop(target->prop,
3275                               target_prop->body.target.target;
3276     }
3277
3278     (void) exists(true_target);
3279
3280     if (true_target->state == build_running) {
3281         return(build_running);
3282     }
3283     if (true_target->stat.time != file_doesnt_exist) {
3284         result = build_ok;
3285     }
3286
3287     /* get line property for the target */
3288     line = get_prop(true_target->prop, line_prop);
3289
3290     /* first check for explicit rule */
3291     if (line != NULL && line->body.line.command_template != NULL) {
3292         result = build_ok;
3293     }
3294     /* try to find pattern rule */
3295     if (result == build_dont_know) {
3296         result = find_percent_rule(target, NULL, false);
3297     }
3298
3299     /* try to find double suffix rule */
3300     if (result == build_dont_know) {
3301         if (target->is_member) {
3302             Property member = get_prop(target->prop, member_prop);
3303             if (member != NULL && member->body.member.member != NULL)
3304                 result = find_ar_suffix_rule(target, member->bod
3305         } else {
3306             result = find_double_suffix_rule(target, NULL, f
3307         }
3308     } else {
3309         result = find_double_suffix_rule(target, NULL, false);
3310     }
3311 }
3312
3313 /* try to find suffix rule */
3314 if ((result == build_dont_know) && second_pass) {
3315     result = find_suffix_rule(target, target, empty_name, NULL, fals
3316 }
3317
3318 /* check for sccs */
3319 if (result == build_dont_know) {
3320     result = sccs_get(target, NULL);
3321 }
3322
3323 /* try to find dyn target */
3324 if (result == build_dont_know) {
3325     Name dtarg = find_dyntarget(target);
3326     if (dtarg != NULL) {
3327         result = target_can_be_built(dtarg);
3328     }
3329 }
3330
3331 /* check whether target was mentioned in makefile */
3332 if (result == build_dont_know) {
3333     if (target->colons != no_colon) {
3334         result = build_ok;
3335     }

```

```

3336     }
3337
3338     /* result */
3339     return result;
3340 }

```

```

*****
37160 Wed May 20 12:01:46 2015
new/usr/src/cmd/make/lib/mksh/macro.cc
make: restore a couple of blocks of code from DISTRIBUTED that should have been
*****
_____unchanged_portion_omitted_____

1045 /*
1046 * We use a permanent buffer to reset SUNPRO_DEPENDENCIES value.
1047 */
1048 char *sunpro_dependencies_buf = NULL;
1049 char *sunpro_dependencies_oldbuf = NULL;
1050 int sunpro_dependencies_buf_size = 0;

1052 /*
1053 * setvar_daemon(name, value, append, daemon, strip_trailing_spaces)
1054 *
1055 * Set a macro value, possibly supplying a daemon to be used
1056 * when referencing the value.
1057 *
1058 * Return value:
1059 *           The property block with the new value
1060 *
1061 * Parameters:
1062 *   name       Name of the macro to set
1063 *   value      The value to set
1064 *   append     Should we reset or append to the current value?
1065 *   daemon     Special treatment when reading the value
1066 *   strip_trailing_spaces from the end of value->string
1067 *   debug_level Indicates how much tracing we should do
1068 *
1069 * Global variables used:
1070 *   makefile_type Used to check if we should enforce read only
1071 *   path_name      The Name "PATH", compared against
1072 *   virtual_root   The Name "VIRTUAL_ROOT", compared against
1073 *   vpath_defined  Set if the macro VPATH is set
1074 *   vpath_name     The Name "VPATH", compared against
1075 *   envvar        A list of environment vars with $ in value
1076 */
1077 Property
1078 setvar_daemon(register Name name, register Name value, Boolean append, Daemon da
1079 {
1080     register Property macro = maybe_append_prop(name, macro_prop);
1081     register Property macro_apx = get_prop(name->prop, macro_append_pr
1082     int length = 0;
1083     String_rec destination;
1084     wchar_t buffer[STRING_BUFFER_LENGTH];
1085     register Chain chain;
1086     Name val;
1087     wchar_t *val_string = (wchar_t*)NULL;
1088     Wstring wcb;

1091     if ((makefile_type != reading_nothing) &&
1092         macro->body.macro.read_only) {
1093         return macro;
1094     }
1095     /* Strip spaces from the end of the value */
1096     if (daemon == no_daemon) {
1097         if (value != NULL) {
1098             wcb.init(value);
1099             length = wcb.length();
1100             val_string = wcb.get_string();
1101         }
1102         if ((length > 0) && iswspace(val_string[length-1])) {
1103             INIT_STRING_FROM_STACK(destination, buffer);

```

```

1104         buffer[0] = 0;
1105         append_string(val_string, &destination, length);
1106         if (strip_trailing_spaces) {
1107             while ((length > 0) &&
1108                 iswspace(destination.buffer.start[length-
1109                     destination.buffer.start[--length] = 0;
1110             }
1111         }
1112         value = GETNAME(destination.buffer.start, FIND_LENGTH);
1113     }
1114 }
1115
1116 if (macro_apx != NULL) {
1117     val = macro_apx->body.macro_appendix.value;
1118 } else {
1119     val = macro->body.macro.value;
1120 }

1122 if (append) {
1123     /*
1124     * If we are appending, we just tack the new value after
1125     * the old one with a space in between.
1126     */
1127     INIT_STRING_FROM_STACK(destination, buffer);
1128     buffer[0] = 0;
1129     if ((macro != NULL) && (val != NULL)) {
1130         APPEND_NAME(val,
1131             &destination,
1132             (int) val->hash.length);
1133         if (value != NULL) {
1134             wcb.init(value);
1135             if (wcb.length() > 0) {
1136                 MBTOWC(wcs_buffer, " ");
1137                 append_char(wcs_buffer[0], &destination)
1138             }
1139         }
1140     }
1141     if (value != NULL) {
1142         APPEND_NAME(value,
1143             &destination,
1144             (int) value->hash.length);
1145     }
1146     value = GETNAME(destination.buffer.start, FIND_LENGTH);
1147     wcb.init(value);
1148     if (destination.free_after_use) {
1149         retmem(destination.buffer.start);
1150     }
1151 }

1153 /* Debugging trace */
1154 if (debug_level > 1) {
1155     if (value != NULL) {
1156         switch (daemon) {
1157             case chain_daemon:
1158                 (void) printf("%s =", name->string_mb);
1159                 for (chain = (Chain) value;
1160                     chain != NULL;
1161                     chain = chain->next) {
1162                     (void) printf(" %s", chain->name->string
1163                 }
1164                 (void) printf("\n");
1165                 break;
1166             case no_daemon:
1167                 (void) printf("%s= %s\n",
1168                     name->string_mb,
1169                     value->string_mb);

```

```

1170         break;
1171     } else {
1172         (void) printf("%s =\n", name->string_mb);
1173     }
1174 }
1175 /* Set the new values in the macro property block */
1176 /**/
1177 if (macro_apx != NULL) {
1178     macro_apx->body.macro_appendix.value = value;
1179     INIT_STRING_FROM_STACK(destination, buffer);
1180     buffer[0] = 0;
1181     if (value != NULL) {
1182         APPEND_NAME(value,
1183             &destination,
1184             (int) value->hash.length);
1185         if (macro_apx->body.macro_appendix.value_to_append != NU
1186             MBTOWC(wcs_buffer, " ");
1187             append_char(wcs_buffer[0], &destination);
1188         }
1189     }
1190     if (macro_apx->body.macro_appendix.value_to_append != NULL) {
1191         APPEND_NAME(macro_apx->body.macro_appendix.value_to_appen
1192             &destination,
1193             (int) macro_apx->body.macro_appendix.value
1194         }
1195     value = GETNAME(destination.buffer.start, FIND_LENGTH);
1196     if (destination.free_after_use) {
1197         retmem(destination.buffer.start);
1198     }
1199 }
1200 /**/
1201 macro->body.macro.value = value;
1202 macro->body.macro.daemon = daemon;
1203 /*
1204  * If the user changes the VIRTUAL_ROOT, we need to flush
1205  * the root package cache.
1206  */
1207 if (name == path_name) {
1208     flush_path_cache();
1209 }
1210 if (name == virtual_root) {
1211     flush_vroot_cache();
1212 }
1213 /* If this sets the VPATH we remember that */
1214 if ((name == vpath_name) &&
1215     (value != NULL) &&
1216     (value->hash.length > 0)) {
1217     vpath_defined = true;
1218 }
1219 /*
1220  * For environment variables we also set the
1221  * environment value each time.
1222  */
1223 if (macro->body.macro.exported) {
1224     static char *env;
1225
1226     if (!reading_environment && (value != NULL)) {
1227         if (!reading_environment && (value != NULL) && value->dollar) {
1228             Envvar p;
1229
1230             for (p = envvar; p != NULL; p = p->next) {
1231                 if (p->name == name) {
1232                     p->value = value;
1233                     p->already_put = false;
1234                     goto found_it;

```

```

1235     }
1236     }
1237     p = ALLOC(Envvar);
1238     p->name = name;
1239     p->value = value;
1240     p->next = envvar;
1241     p->env_string = NULL;
1242     p->already_put = false;
1243     envvar = p;
1244 found_it:;
1245     } if (reading_environment || (value == NULL) || !value->dollar)
1246     } else {
1247         length = 2 + strlen(name->string_mb);
1248         if (value != NULL) {
1249             length += strlen(value->string_mb);
1250         }
1251         Property env_prop = maybe_append_prop(name, env_mem_prop
1252         /*
1253          * We use a permanent buffer to reset SUNPRO_DEPENDENCIE
1254          */
1255         if (!strcmp(name->string_mb, NOCATGETS("SUNPRO_DEPENDEN
1256             if (length >= sunpro_dependencies_buf_size) {
1257                 sunpro_dependencies_buf_size=length*2;
1258                 if (sunpro_dependencies_buf_size < 4096)
1259                     sunpro_dependencies_buf_size = 4
1260                 if (sunpro_dependencies_buf)
1261                     sunpro_dependencies_oldbuf = sun
1262                 sunpro_dependencies_buf=getmem(sunpro_de
1263             }
1264             env = sunpro_dependencies_buf;
1265         } else {
1266             env = getmem(length);
1267         }
1268         env_alloc_num++;
1269         env_alloc_bytes += length;
1270         (void) sprintf(env,
1271             "%s=%s",
1272             name->string_mb,
1273             value == NULL ?
1274             "" : value->string_mb);
1275         (void) putenv(env);
1276         env_prop->body.env_mem.value = env;
1277         if (sunpro_dependencies_oldbuf) {
1278             /* Return old buffer */
1279             retmem_mb(sunpro_dependencies_oldbuf);
1280             sunpro_dependencies_oldbuf = NULL;
1281         }
1282     }
1283     if (name == target_arch) {
1284         Name ha = getvar(host_arch);
1285         Name ta = getvar(target_arch);
1286         Name vr = getvar(virtual_root);
1287         int length;
1288         wchar_t *new_value;
1289         wchar_t *old_vr;
1290         Boolean new_value_allocated = false;
1291
1292         Wstring ha_str(ha);
1293         Wstring ta_str(ta);
1294         Wstring vr_str(vr);
1295
1296         wchar_t * wcb_ha = ha_str.get_string();
1297         wchar_t * wcb_ta = ta_str.get_string();
1298         wchar_t * wcb_vr = vr_str.get_string();

```

```
1300     length = 32 +
1301         wslen(wcb_ha) +
1302         wslen(wcb_ta) +
1303         wslen(wcb_vr);
1304     old_vr = wcb_vr;
1305     MBSTOWCS(wcs_buffer, NOCATGETS("/usr/arch/"));
1306     if (IS_WEQUALN(old_vr,
1307                   wcs_buffer,
1308                   wslen(wcs_buffer))) {
1309         old_vr = (wchar_t *) wschr(old_vr, (int) colon_char) + 1
1310     }
1311     if ( (ha == ta) || (wslen(wcb_ta) == 0) ) {
1312         new_value = old_vr;
1313     } else {
1314         new_value = ALLOC_WC(length);
1315         new_value_allocated = true;
1316         WCSTOMBS(mbs_buffer, old_vr);
1317         (void) wsprintf(new_value,
1318                        NOCATGETS("/usr/arch/%s/%s:%s"),
1319                        ha->string_mb + 1,
1320                        ta->string_mb + 1,
1321                        mbs_buffer);
1322     }
1323     if (new_value[0] != 0) {
1324         (void) setvar_daemon(virtual_root,
1325                             GETNAME(new_value, FIND_LENGTH),
1326                             false,
1327                             no_daemon,
1328                             true,
1329                             debug_level);
1330     }
1331     if (new_value_allocated) {
1332         retmem(new_value);
1333     }
1334 }
1335 return macro;
1336 }
_____unchanged_portion_omitted_____
```