Web-SynDic

Web System for Demonstrating the Syntactic Algorithms for Solving Linear Equations in Nonnegative Integers (Nonnegative Linear Diophantine Equations)

Design
Iteration 2.0 (started August 2004)

Department of Computer Science, Petrozavodsk State University, Russia

25th November 2004

Contents

1 External algorithms 2
1.1 Generators . . . 2
1.2 Solvers . . . 2
1.2.1 `/proc` file system . . . 2
1.2.2 Functions like `getrusage()` . . . 3
1.2.3 Kernel module . . . 3
1.3 Action points . . . 3
1.4 Scheme for external algorithm control . . . 4
1.5 Scheme for memory usage monitoring . . . 4

2 User Interface 5
2.1 User Limits . . . 5
2.2 Generator Parameters . . . 5
2.3 Password checking . . . 6

3 Web server 6
3.1 Class SessionManager . . . 6
3.2 Class SessionMonitor . . . 6

4 Algorithm server 9
4.1 Class SolverSpooler . . . 9
4.2 Class GeneratorSpooler . . . 10
4.3 Class CheckSolutions . . . 11

5 Management 11
5.1 Class Management . . . 11
5.2 Class UserProfile . . . 12

1 External algorithms

1.1 Generators

It reasonable to use the approach similar to measuring resources for solvers. Unified technology is simple for check and modification if needed. Also for generators the measuring process can be made simpler by checking only restrictions for CPU work time and memory usage.

1.2 Solvers

We considered the following three approaches for measurement of solver resource usage.

1.2.1 `/proc` file system

Description: The `/proc` file system feature gets all required information for measuring resources. This is the simplest approach to monitor a process. This approach was used in the previous (first) version of Web-SynDic.

Disadvantages:

— The refresh time is 0.01 second (one CPU jiffy). Sometimes it is too big.

— The obtained information may be incorrect. For example CPU work time may be greater than real work time (asynchronous update).

1.2.2 Functions like `getrusage()`

Description: This is a better variant because the program is portable between different Unix systems.

Disadvantages:

— Some versions of OS Linux do not support all features of this functions. For example `getrusage()` at Linux kernel 2.4 support only real work time and CPU work time.

— The obtained information may be incorrect. For example CPU work time may be greater than real work time (asynchronous update).

1.2.3 Kernel module

Description: This variant gets full real time information about monitored processes.

Disadvantages:

— For such working with OS kernel the program needs root rights.

— This approach decreases Linux protection and needs very careful implementation and corresponding verification.

1.3 Action points

After the team seminar with our system Administrator we produce the following directions to consider further.

— Check `getrusage()` on our Linux systems. Result: Current versions of Linux kernel do not full support this function.

— Check assumption that size of dynamic memory for a process is always non-decreasing. Result: Unfortunately, size of dynamic memory may increase and decrease periodically.

— Is it reasonable to use the `/proc` file system feature? Result: Yes. The measuring algorithm should use resources very carefully (low consumption).

— Is it reasonable to use `ulimit` for defining time and memory limits for a process? Result: Yes. The time restrictions should be set in seconds. When the CPU work time limit exceeded the program terminated with exit code 152 — "CPU time limit exceeded". In case of memory limit exceeding the program terminated with exit codes 139 — "Segmentation fault" or 127 — "Error while loading shared library" or own exit code (e.g. 1).

— When measure the elapsed time? Result: CPU work time is determined after process termination or when the process has became a zombie.

— Is it worth to use the lib.c library (memory allocation)? Result: No, there is too complicated to use this library.

1.4 Scheme for external algorithm control

Require: An external algorithm to run (solver or generator).
Ensure: the external algorithm has run and terminated, its resource consumption was measured.

1. Set memory and CPU work time restrictions (`ulimit` program or `setrlimit()` function) for the process.

2. Start the external algorithm (`exec()` and `fork()` functions).

3. Monitor *the memory consumption* of the algorithm (use `/proc` file system, the monitoring must be careful and accurate).

4. Get exit code and measure *the time consumption* when algorithm has terminated (use `wait4()` function);

Algorithm server returns exit code and the resource usage measurement.

1.5 Scheme for memory usage monitoring

The delay between two consequent measurements is calculated using the following rules.

— Minimal delay is 0.5 CPU jiffy ($5 \cdot 10^6$ ns).

— Initially the current delay is equal to minimal one; the next 3 measurements do not change the delay.

— When a measurement shows a greater (significantly greater?) value of memory usage the current delay is set to minimal one; the next 3 measurements do not change the delay.

— When a measurement shows the same (almost the same) value of memory usage as it was in the previous measurement, the current delay is increased by 2.

— When a measurement shows less memory usage than previous one, the current delay is not changed.

— If a new value for the delay becomes more than 1% of time limit for the external algorithms, then the current delay is not changed.

— The real delay is calculated as the current delay value minus a random value in range from 0 to 50% of the current delay.

2 User Interface

2.1 User Limits

Only the following fields are available in the "User Limits" form for unprivileged users:

- max. time for solving,
- max. memory for solving,
- max. number of basis solutions for ANLDE system,
- max. number of basis solutions in a report.

Other fields (input data constraints) are only available for the administrator.

References on user requirements

- "User Limits management" (A.1).

2.2 Generator Parameters

Fields for the following parameters for the generation are added to the "Process an ANLDE System" and "Process a Set of ANLDE Systems: Generate" forms:

- max. values of coefficients for ANLDE system,
- max. values of coefficients for basis solutions,
- number of equations in ANLDE system,
- number of unknowns in ANLDE system,
- number of ANLDE systems in a set (only for "Process a Set of ANLDE Systems" form),
- max. number of basis solutions for ANLDE system.

References on user requirements

- "User Limits management" (A.1).

2.3 Password checking

The following changes are added to the "User information" form.

- "Change password" checkbox is replaced on "Change password" button.
- "Old password" text field is added in Change password group.
- "Remove account" checkbox is replaced on "Remove account" button.
- "Old password" text field is added in Remove account group.

References on user requirements

- "User passwords" (A.9).

3 Web server

3.1 Class SessionManager

Methods:

- `public static UserProfile getUserProfileFromSession(HttpSession session)`
  This function should return `UserProfile` object bound to *session*.

  Arguments:

  - `HttpSession session` represents the session containing `UserProfile` object.

  Returns:

  `UserProfile` object bound to *session*, if it exists; `null`, otherwise.

References on user requirements

- 'Server load information' (A.3).

3.2 Class SessionMonitor

Class `SessionMonitor` (see Fig. 1) is used to store information about active sessions in the web system including registered users, anonymous users, and all active users.

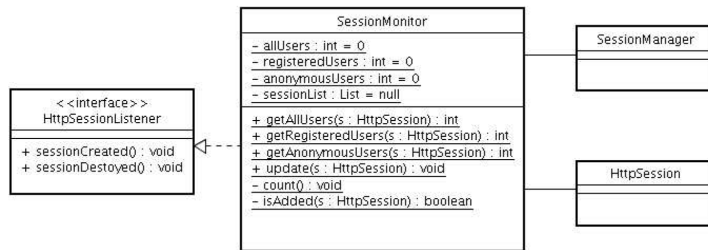

Figure 1: Class SessionMonitor

Fields:

- `private static int allUsers`

  `private static int registeredUsers`

  `private static int anonymousUsers`

  The fields contain numbers of all active users, registered users, and anonymous users in the web system.

- `private static List sessionList`

  Contains list of active sessions represented by `HttpSession` objects.

Methods:

- `public static int getAllUsers(HttpSession session)`

  `public static int getRegisteredUsers(HttpSession session)`

  `public static int getAnonymousUsers(HttpSession session)`

  The functions should return corresponding numbers of all active users, registered users, and anonymous users in the web system. The functions should implement the following algorithm:

  1. If *session* is not presented in *sessionList* (`isAdded()` returns `true`), call `count()` to evaluate new numbers of users.
  2. Return number of corresponding users.

  Arguments:

  – `HttpSession session` represents the session containing `UserProfile` object.

- `public static void update(HttpSession session)`

  This function is used to update information about all active (registered and anonymous) users in the web system. This function should be called by `SessionManager` objects after binding `UserProfile` objects to the session (and thus changing number of corresponding users). It should call `isAdded()` and mandatory `count()` functions.

  Arguments:

  – `HttpSession session` represents the session containing `UserProfile` object.

- `private static void count()`

  This function is used to count values of users information fields. It should implement the following algorithm:

  1. Set *allUsers*, *registeredUsers*, and *anonymousUsers* to zero.
  2. Get next `HttpSession` object from *sessionList* and extract `UserProfile` object using `SessionManager.getUserProfileFromSession()` function (section 3.1).
  3. If `UserProfile` object method `canChangeProfile()` returns true, then *registeredUsers* value is increased; otherwise, *anonymousUsers* value is increased.
  4. Increase *allUsers* value.
  5. If *sessionList* contains more items goto step 2.

- `private static boolean isAdded(HttpSession session)`

  This function should create *sessionList* if it is not yet initialized, and add *session* object to *sessionList* if it is not already presented there.

  Arguments:

  – `HttpSession session` represents the session containing `UserProfile` object.

  Returns:

  `true`, if *session* is added to *sessionList*; `false`, otherwise.

- `public void sessionCreated(HttpSessionEvent event)`

  This is empty function used for `HttpSessionListener` interface realization.

  Arguments:

  – `HttpSessionEvent event` contains `HttpSession` object representing created session.

- `public void sessionDestroyed(HttpSessionEvent event)`

  This function is called automatically on session invalidation. It should remove `HttpSession` object from the list of active sessions *sessionList* and mandatory call `count()` function.

Arguments:

– `HttpSessionEvent event` contains `HttpSession` object representing destroyed session.

Implementation notes

- The class `SessionMonitor` should be configured in the web application deployment descriptor to accept notifications about created and destoyed sessions. The following information should be added to the `web.xml` file:

```
<listener>
  <listener-class>
  ru.petrsu.websyndic.webserver.SessionMonitor
  </listener-class>
</listener>
```

  After that an instance of SessionMonitor object would be automatically created by Tomcat web server on application installation.

- The method `update()` should be called every time after binding new `UserProfile` object to the session to keep correct information about active users (see Fig. 2).

References on user requirements

- 'Server load information' (A.3).

4 Algorithm server

4.1 Class SolverSpooler

Methods:

- Method:

  `public int getSolverTasks()`

  This function returns the number of tasks in the solver spooler queue.

  Returns:

  the number of tasks in the solver spooler queue.

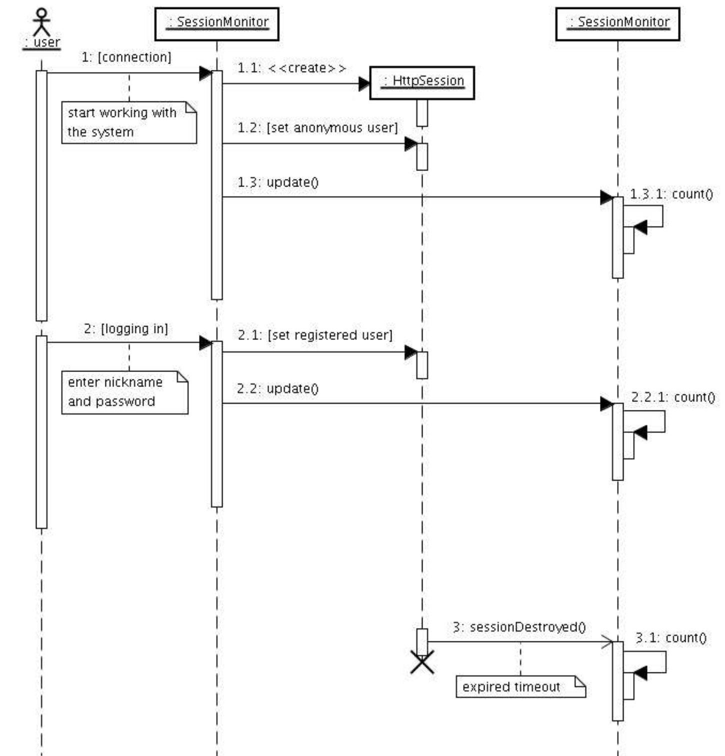

Figure 2: Keeping information about active sessions up-to-date

References on user requirements

- 'Server load information' (A.3).

4.2 Class GeneratorSpooler

Methods:

- Method:

  `public int getGeneratorTasks()`

  This function returns the number of tasks in the generator spooler queue.

Returns:

the number of tasks in the generator spooler queue.

References on user requirements

- 'Server load information' (A.3).

4.3 Class CheckSolutions

Methods:

- Method:

  `public boolean isANLDESolution(ANLDESystem anlde, Solution sol)`

  This function checks if *sol* is a solution of *anlde*. The function should implement the following algorithm:

  1. Get next solution from *sol*.
  2. If after substitution the solution doesn't match, return `false`.
  3. If there are more candidate solutions goto step 1.
  4. Return `true`.

  Arguments:

  - `ANLDE anlde` represents the ANLDE system.
  - `Solution sol` represents the candidate solution of given ANLDE system.

  Returns:

  `true`, if *sol* is the solution of *anlde*; `false`, otherwise.

References on user requirements

- 'Check solutions' (A.9).

5 Management

5.1 Class Management

Methods:

- Method:

  `private static String hex ( byte[] array )`

  This function convert byte sequence in the hex format (32 bytes)

  Arguments:

  - `byte[] array` is an encrypted byte sequence.

- Method:

  `static String md5 ( String message )`

  This function encryptes user password using md5 algorithm.

  Arguments:

  - `String message` is a user password.

- Method:

  `public static boolean checkOldPassword ( String old_password, String current_password )`

  This function check old user password.

  Arguments:

  - `String old_password` is a user password.
  - `String current_password` is a checked password.

References on user requirements

- 'User passwords' (A.9).

5.2 Class UserProfile

Methods:

- Method:

  `public void setEncryptedPassword (String password)`

  This function sets the encrypted password.

  Arguments:

  - `String password` is a user password.

References on user requirements

- 'User passwords' (A.9).

Implementation notes:

`getPassword` method will be renamed to `getEncryptedPassword`.