2020 10th IEEE International Conference on Control System, Computing and Engineering (ICCSCE2020), 21–22 August 2020, Penang, Malaysia

Pushdown Automata Versioning Framework for Web-Based Developmental Surveillance Systems

Yan Huan Ch'ng
School of Computer Sciences
Universiti Sains Malaysia
Gelugor, Penang, Malaysia
chng_yan_huan@student.usm.my

Mohd Azam Osman
School of Computer Sciences
Universiti Sains Malaysia
Gelugor, Penang, Malaysia
azam@.usm.my

Hui Ying Jong
School of Humanities
Universiti Sains Malaysia
Gelugor, Penang, Malaysia
jonghuiying@usm.my

***Abstract*—The development of existing software system, including web-based developmental surveillance programs, adopt linear software versioning systems wherein all files associated to a particular software version is pre-packed together and distributed to clients. In this paper, the two inconveniences incurred by linear software versioning onto web-based software development have been highlighted in terms of the software system update and rollback infrastructure. The use of a new Pushdown Automata (PDA) - based versioning framework has been proposed to resolve these issues, by reducing update file storage requirements on the master server, and by simplifying the update files fetching process and version update/rollback installation process. Based on the proposed PDA-based versioning framework, a description is provided on how branched software versioning systems can be implemented to facilitate the development of a light-weight central repository server which allows access to different online language screening tools for all respective needs.**

Keywords—Language screening tool, web-based software, branched software versioning, software update, rollback, pushdown automata

I. Introduction

In computer systems involving master servers which moderate and control different software versions on their clients, the three main tasks being performed include software version update, software version verification and software version rollback. Likewise, multi-version web applications or web-based tools which reside on the host server go through the same phases when accessed through a domain and loaded onto client devices. Upon a software version update request, the software files residing in client side, which are deemed to be outdated by the master server, are deleted and replaced by updated files. In the case of software rollback, files are deleted and replaced by an older version of the same file. This bears similarity to undo/redo (UR) operations in most computer systems, which are more commonly linear than not. As a result, software updates and rollbacks via existing common version control systems rarely stores multiple historical states of one single version. Instead, each software version is treated as a single state, in which updating or rollback moves the software version linearly along plausible states (versions) connected as a single line. In this paper, the use of pushdown automata (PDA) has been proposed to emulate version control capabilities observed within non-linear UR models, allowing software system version update and rollback procedures to be represented as a PDA state diagram instead of a fixed stack or linear list.

A. Scope of Research

This paper was written as a sub-study of an ongoing research involving the development of a web-based language screening tool which offers Specific Language Impairment (SLI) diagnosis and music therapy, known as the Psychology Software Tool (PST). A survey conducted on existing web-based language screening tools, which are meant to be used by speech-language therapists to screen patients for developmental language disorders, reveal that most of these web-based diagnosis tools are subjected to inconveniences incurred by the linear model of multi-version software development. As a result, these systems which offer speech-language pathology services suffer from higher development costs, slow file server respond times, reduced service choices, and more [11]. With respect to the data that has been collected in the survey, as well as the affiliation to the original research on web-based language screening tools, the PDA-based versioning framework delineated in this paper has been studied solely within the context of web-based developmental surveillance systems. However, the actual application of this proposed conceptual framework should be feasible over the span of most computer systems, wherever client-server system architecture is involved.

B. Overview of multi-version software systems

When software developers manage multiple software versions and editions, software versions are commonly linear, e.g. version 1.0 precedes 1.1, which precedes 1.2, so on and so forth. The installation process of a particular computer system which follows this linear model identifies the target software version, which informs the system regarding the dependencies required for installation of that particular software version, and proceed to collect the files needed to update the existing software. In many cases, the old version of the software has to be removed before installation of an updated version of the same software takes place. In other cases, minor patches are introduced to the existing system as minor updates.

Take operating system updates as an example, quality updates involve minor changes and bug fixes, while feature updates cause the OS to make a backup of itself, uninstalls the existing version of the OS and replace it with the new build containing the major version update. Likewise, rolling back to an older major build uninstalls the latest feature update to go back to an older version by restoring the backup of the OS. As such, software versions handled by such versioning systems can be represented as an ordered list, which users can choose to move back and forth on states within the list. The most basic way to implement this would be to store the exact state of the software version, i.e. all associated files, in these states which make up the ordered list according to their version number. This is simple to implement and is widely used, but introduces a number of issues which, may not bother most users, but exist nonetheless. One of these problems involve storage requirements, in the sense that the files relevant to a particular version may be largely duplicated, which as a

Authorized licensed use limited to: UNIVERSITY OF CONNECTICUT. Downloaded on November 16,2020 at 16:26:38 UTC from IEEE Xplore. Restrictions apply.

result causes the host server home to a great deal of redundancies. For instance, version 1.0 contains files A, B, C and D, while version 1.1 contains files A, B, C, D and E. To us, it is obvious that the first four files are duplicated, with E being the sole addition to the software system.

Given the basic implementation described early, however, such redundancies are overlooked. Excluding minor patches, software systems containing many different major versions will suffer from this implementation in the sense that the storage required to hold all files in all states within the version list may grow exponentially, or at best linearly. To conclude the first issue, it is plausible to assume that linear versioning is memory intensive. Another inconvenience posed by the basic implementation involves its linearity, and how it prevents software system developers from providing different branches of the same version of the software system. Figure 1 below shows an example of branched software versioning.

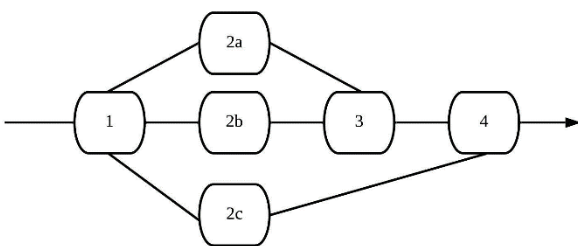

Fig. 1. Branched Software Versioning

In Figure 1 above, observe that version 1 of the software system branches off into three updates on the same timeline, namely versions 2a, 2b and 2c. What this represents is a scenario wherein these three updates are “sister versions”, versions which do not precede one another, but are pushed out in parallel to the user base. In the context of developmental language screening tools, these sister versions may represent the same software system, which offers different diagnosis and treatment services. For instance, version 2a may contain updates which introduces sentence-picture matching task for SLI diagnosis, 2b for speech repetition tests, while 2c is a debug build which focuses on improving version 1 offline accessibility options. Neither version of the 2x series is an improvement in terms of SLI screening capabilities over the other, but rather an update branch which speech-language pathologists can choose from - Therapists which are more focused on offline work and require the corresponding tools can choose to go for 2c, while others utilizing the software may go for 2a or 2b, depending on other language screening tools that they use in conjunction to the target software. Observe that version 2a and 2b converges into the next mandatory update to version 3, whereas users on 2c can bypass version 3 to go for version 4. In this case, version 3 may contain fixes which address certain network issues, and this is not required for 2c users since the changes made in 2a and 2b are not present in 2c. In terms of rollback, observe that clients running on version 4 can choose to either rollback to version 3 or version 2c in a single step. The amount of software system versioning flexibility is not seen within linear versioning systems or frameworks. In order to work around this issue, existing software system developers have to launch multiple versions of the same software system under alternate prefix or postfix titles. As a matter of fact, branched versioning systems or tools have been introduced in the past, but never gained mainstream popularity due to implementation complexity and the relative simplicity offered by linear versioning systems. Furthermore, the storage issues previously mentioned undermines the feasibility of version branching. To summarize the issues and inconveniences incurred by the common version update and rollback systems:

- File redundancy across multiple versions of the same software system causes existing update and rollback systems to be memory intensive.
- The lack of version branching in linear versioning systems hinders the ability of developers to design flexible software in a way which allows users to update and rollback to desired software versions.

In the next section of this paper, the existing systems and related work is reviewed. This review reveals past research on measures to overcome these issues, and feasible results which can be leveraged to develop an innovative update and rollback framework involving the use of PDA.

II. Background and Related Work

A. Pushdown Automata

Pushdown Automata are finite state machines (FSM) equipped with a stack (pushdown stack), and the corresponding ability to push and pop stack symbols on each state transition. The transition from a particular state to another state in pushdown automata can depend not only on the input symbol to a particular state, but also the stack symbol which is currently on top of the stack. A PDA differs from a FSM mainly in the aspect of memory - FSMs are not equipped with structures to store the previous states in which state transition has taken place in, while the stack of a PDA can play the role of external memory.

B. Literature Review

Over the years, a number of research studies have been conducted on the involvement of automata theory and its practicality within undo/redo functions of computer software [10]. Takagi et al. [2] proposes the use of PDA for modelling and testing complex UR functionality within software, and defined the scope of UR elements which can be represented or described by a PDA. As such, the work by Takagi et al [2] became the theoretical foundation of this paper - It is hypothesized that software update and rollback functionality is similar to transitions in UR, which, in their work, are represented based on the symbols within the PDA stack [2]. According to [2], a usual PDA has only one stack, but modeling UR functions requires stacks to hold a history of state transitions for the undo function and the redo function respectively. However, the possibility of simplifying the use of PDA to the default single-stack architecture to simulate UR functionality has not been thoroughly explored. Since software update and rollback are relatively older topics within the field of computer sciences, a number of older journal articles also provided relevant insight into the implementation of these functionality. Such work include Chandy K. M. et al. [3], which depicts how system checkpoints are created and stored on the system, and the aspects which may trigger an automatic rollback procedure. According to Chandy, checkpoints (which, in the context of this paper, can be considered as older versions of a system

Authorized licensed use limited to: UNIVERSITY OF CONNECTICUT. Downloaded on November 16,2020 at 16:26:38 UTC from IEEE Xplore. Restrictions apply.

software) are chronological records of all data and the transactions involved in the system, at a particular point of time [3]. During the year of when [3] was published (1975), it was clearly stated that partial software recovery and rollback not only takes a significant amount of time due to check-pointing, but also incurs high hardware cost for storage of redundant data [3]. Out of the three rollback and recovery (RR) models proposed in [3], neither involves the use of PDA to describe backup data storage nor to implement rollback functionality. However, the graph theory was used to describe model C, wherein the number of arithmetic operations required to compute optimal checkpoints can be calculated [3]. A more recent study describes methods to optimize rollback and re-computation costs within workflow management systems [4]. Workflow management systems execute long-running computational and data-intensive pipelines of operations [4]. Workflow rollback and re-computation tasks are in huge contrast against rolling back software systems, wherein the latter deals with deleting or replacing files with an older counterpart, while the former has to account for checkpoints made during the middle of active computations [4]. That said, the paper provided plenty insight into how rollback, while not maintaining any execution state of processes, allows faster system restoration due to the lack of need for error-checking and re-computation [4]. The use of versioning filesystems with continuous snapshot ability was also explored, and was said to provide efficient rollback capabilities [4]. Likewise, new approaches involving the use of an extra recovery layer containing order tables for rollback purposes were proposed in the field of distributed systems [5][8]. In terms of software updates within the field of IoT, several research have delineated the overall architecture and software components of the IoT stack, including the few aspects which have to be considered when updating software, such as inter-module compatibility, network compatibility and platform compatibility [6]. Due to how the software stack and the corresponding updates are designed, some update scenarios involve the replacement of the entire code base, in which case compatibility analysis would be crucial prior to executing an update process [6]. In other cases, updating application-level code blocks is said to allow the remaining software components to stay intact [6]. In terms of future work, it has been mentioned that code modularity and code isolation practices are crucial towards increasing the security and efficiency of software system updates within the field of IoT [6][7].

On the other hand, research studies within the context of web-based language screening tools and developmental surveillance systems have commonly focused on evaluating the effectiveness and reliability of these tools and systems when being delivered over the internet [11]. Bandwidth or network connection issues have commonly been reported as an issue impacting the accuracy of online-based tests, since latency not only causes audio/video distortion, but also affects timing in tests which contain time-sensitive elements [11]. It is pointed out that the practicality of diagnosis conducted over the internet through web browsers lies on two extremes, either of high practicality through cost reduction and increased diagnosis effectiveness [12][14], or impracticality due to a lack of facility [12]. In the general sense, the performance of web-based tests rely greatly on the existing computer network which they run on [12]. This heavy reliance hints towards the importance of implementing a fast, secure and cost-effective distribution infrastructure for web-based language screening tools, by considering different aspects, from how files are stored and retrieved from the file system, all the way to the capabilities of the physical web server used to host the tools [13].

III. Proposed Solution

With respect to the inconveniences of common software system update and rollback infrastructure described in Section I, the Pushdown Automata (PDA) Versioning Framework for web-based developmental surveillance systems is proposed, with the following objectives in mind:

- Reduce hardware storage space requirements of the master server which stores the software version base.
- Offer branched versioning system to software system developers.

In order to explain our proposed framework with relative ease, Figure 1 shall be referred to for an example of branched software versioning as a potential case study, in which versions 1, 2a, 2b, 2c, 3, and 4 are present and interconnected. Assume that a collection of files are tied to each of the distinct software system versions. For instance, version 1 includes files A, B and C; version 2 includes files A, B, D and E, etc.

The following sections discuss the involvement of PDA to achieve each of the two objectives being mentioned, and will discuss these few examples in detail.

A. PDA-based Update Fetching Component

As briefly mentioned in the introduction of this paper, existing software system update files are mostly implemented by packing all relevant files into an installer, which can then be pushed to client side for installation to take place. For large systems, this means that installers take up a huge space within the master server. The changes in between neighboring versions are small, and most of the code and files are duplicated. The current PDA framework being proposed takes advantage of the fact that files which span across multiple distinct versions are in fact the same file, and can actually share a common storage space, much like global variables in programming. Instead of copying the file into different version folders, and then packing the individual folders into installers, it is visualized that all files can be simply stored in a fixed partition, and linked to specific software versions through the use of context-free grammar (CFG) designed as PDAs. When an update of the a particular version based on specific parameters (input version string) is requested by the client, the master server consults the “Linker PDA” specific to that version, and generates the installer dynamically by pulling the files required into a temporary folder, which is then packed and sent to the client. Consider the following CFG grammar for a simple file linking rule in a single software system version:

$$S \rightarrow AB$$
$$A \rightarrow w \mid \varepsilon$$
$$B \rightarrow xy \mid z$$

By the definition of our PDA framework and file linking rules, the software system version requires two system components, A and B. The former component, A, is optional, and as such, file ‘w’ which corresponds to component A can either be fetched, or omitted, according to user preference.

Authorized licensed use limited to: UNIVERSITY OF CONNECTICUT. Downloaded on November 16,2020 at 16:26:38 UTC from IEEE Xplore. Restrictions apply.

On the other hand, installation of component B is mandatory, and the files which have to be fetched for installation is either files x and y, or file z. Again, this choice would be provided by the software system developer, and selected by the user. Figure 2 shows the Linker PDA which corresponds to the CFG grammar listed earlier.

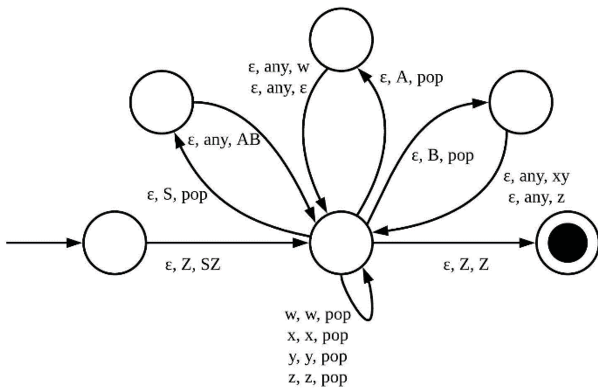

Fig. 2. Linker PDA for file linking rules

According to the proposed framework, all basic update file units reside in a fixed partition in the master server. In the example provided, this includes files w, x, y and z. According to user selection through whatever provided interface which talks to the master server to request for updates, the specific update or rollback version and the required components are selected. The master server consults the Linker PDA, and whenever the Linker PDA pops a stack symbol which corresponds to a file in the partition, the master server fetches the files, which can then be packed and sent to the client. It is also completely possible to go with a more naive implementation of the proposed PDA-based framework, by having a single-line CFG grammar for direct file linking:

$$S \rightarrow abcdef...$$

In the case above, the collection of files linked to the described version are directly fetched upon request, without any optional or alternative components available for user selection.

The proposed framework, however, has yet to address update and rollback scenarios wherein changes to the software system not only include additional files which have to be downloaded, but also obsolete files which may need to be replaced or deleted from the software directory. In order to handle such cases, the proposal of the PDA-based framework can be extended to cover software system file cross-validation functionality, wherein an input string based on the existing files in the target software system is built and fed to a PDA. The functionality of this additional PDA can be extended to include file deletion states. This means that two PDAs are involved: The first Linker PDA resides in the master server, and is consulted whenever an update or rollback request is received; The second PDA is a copy of the first PDA with additional file deletion functionality, which is sent to the client along with the collected update files, and executed in a similar manner to how an installer would be. In short, the second PDA is an installer which installs the files fetched by the Linker PDA. For the rest of this paper, the second PDA is addressed as the "Installer PDA". To describe this in detail, consider Figure 3 below, which shows how the installation process on client-side works via the Installer PDA.

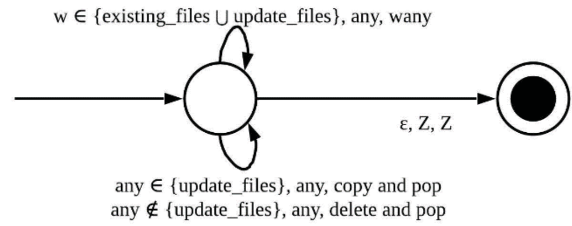

Fig. 3. Client-side Installer PDA with file replace and deletion functionality

The input to the Installer PDA shown in Figure 3 above is a complete string representing the union of existing files in the software system directory and the update files fetched from the server. The Installer PDA attempts to capture input symbols/files which are foreign to the collection of update files, and deletes the associated files. Consider the example wherein the existing software system directory contains files a, b, c and d, while the updated version contains files a, b, c and e. The symbol string resulting from the union of existing files and update files would be "abcde", all of which would be pushed into the PDA stack. After this process, the installer PDA begins to handle file replacement and deletion functionality, where files which are elements of the update files would be copied into the software system directory, either adding to the collection of files there, or by replacing existing files with the same filename. As for file symbols in the stack which are not elements of the update files collection, these files are directly deleted from the software system directory. Both file addition/replacement and deletion steps are accompanied by popping the top of the stack, as such, the Installer PDA will always enter the accepted state (end by empty stack).

B. PDA-based Non-Linear Versioning System

In terms of building a Non-Linear or branched versioning system as opposed to the commonly practiced linear versioning system, the same principles of PDAs and CFG grammar for update and rollback rules can be implemented to regulate version change eligibility. This is, however, optional - There are two scenarios to be contemplated. The first scenario is where there are no boundaries as to which version can be updated or rolled back to another version. The second scenario is where software system version updates and rollbacks are limited to specific rules set by the software system developers. According to the Update Fetching component in our PDA-based framework, software versions are, by default, structured in a way where no rules exist (First scenario) - A client running version 1 or a particular software system implemented according to our framework can perform an update to jump to version 10, skipping across multiple versions in between. This is because file dependencies can be resolved directly via the Installer PDA. As such, the framework already grants freedom from the linearity of common software versioning systems. This can be achieved because software versions are no longer a fixed collection of files, let alone a patch containing only a limited collection of

Authorized licensed use limited to: UNIVERSITY OF CONNECTICUT. Downloaded on November 16,2020 at 16:26:38 UTC from IEEE Xplore. Restrictions apply.

files which may lack dependencies in versions which are not targeted for that particular bug-fix update. The PDA-based framework redefines a software version as a set of rules which can be interpreted by a PDA, in order to, upon demand, dynamically fetch and create an all-compatible version update installer.

In the case where the second scenario is preferred, an additional PDA which resides on the master server has to be created to validate the selected update or rollback path requested by the client. In this case, a CFG grammar can be written, containing the rules of which a single version can transition to. For the rest of this paper, this third PDA is addressed as the "Validator PDA". Figure 4 below shows a general example of what the version eligibility Validator PDA would look like.

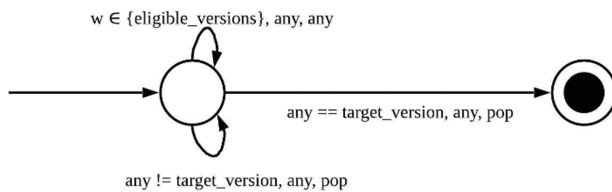

Fig. 4. Update/Rollback target version eligibility Validator PDA

Although the structure of the Validator PDA appears to be similar to that of the Installer PDA, a couple of semantic difference exist. The Validator PDA does not go with acceptance on empty stack, but rather accepts upon popping a stack symbol which corresponds to the target version that the client wishes to update or rollback to. In order for the Validator PDA to work, the master server has to determine the current version of the software system that the client is using, and derive the list of eligible software system versions that the client can transition to. Taking the branched software versioning system from Figure 1 as an example, if the user is currently running version 2a, they are only allowed to rollback to version 1, or update to version 3. As such, the Validator PDA will push 1 and 3 into the PDA stack. The PDA then pops all stack symbols until the target version that the clients wishes to update or rollback to, is found. If the user selects version 3, the Validator PDA moves to the accepted state upon popping stack symbol 3. If the user selects version 4, the PDA pops everything from the stack and does not reach the accepted state, hence rejecting the selection. It is worth noting that the Validator PDA shown in Figure 4 lacks the procedures which reflect deriving eligible versions akin to CFG grammar derivation. This part was left out intentionally, in order to focus on the latter, core functionality of the Validator PDA. Through the reliance on input values from the client for target version, as well as the master server for reading client software system current version, it is shown that the Validator PDA can be used to validate all version update and rollback rules. As such, multiple Validator PDAs are not needed to validate multiple versions, reducing implementation complexity.

C. *Language Screening Central Repository Server*

The PDA-based Update Fetching component and the Non-Linear Versioning System which have been delineated in earlier sections describe the use of a total of three PDAs, namely, the Linker PDA, Installer PDA and Validator PDA. With respect to our base study involving developmental surveillance systems, the PDA-based versioning framework can be applied to develop a central repository server which houses many distinct language screening tools. Figure 5 shows the overall system architecture of the proposed language screening central repository server.

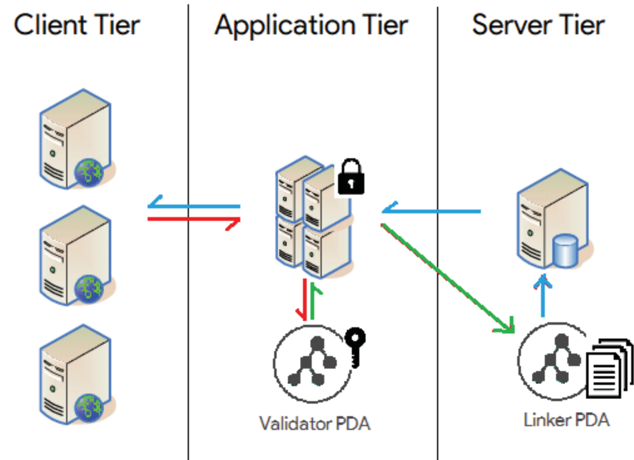

Fig. 5. Central Repository Server System Architecture

Figure 5 shows an implementation of the central repository server involving a three-tier client-server architecture. Red arrows represent client requests, green arrows represent validation flags sent out by the Validator PDA, while blue arrows represent the software files and the corresponding Installer PDAs being sent from the server to the client. The client tier involves the UI and presentation of the application to speech-language therapists who wish to access language screening tools stored in the central repository. The application tier handles the business logic of the repository, acting as an interface between the client and low-level access to files stored on the physical server. The server tier is the backend in which the actual database operates. The server tier contains methods to manipulate files and other information hosted on the server. With respect to the PDA-based versioning framework and the functionality of the separate tiers within the system architecture, the Validator PDA and the Linker PDA are situated at the application tier and server tier respectively. The application server and its Validator PDA forms a lock-and-key model, wherein client requests for a particular language screening tool goes through the Validator PDA. The Validator PDA then decides, based on client request information and custom rules, whether to grant or reject the request. Upon successful validation, the flag triggers the Linker PDA, which starts collecting all the relevant files required to form the target language screening tool, and the corresponding Installer PDA required to unpack and replace the files at client side. The files relevant to the client request are then sent from the server tier to the application tier to be packaged into a single deliverable of appropriate format if required, before being sent to the client.

A few development aspects should be taken into consideration during actual implementation of both the PDA-based framework and system architecture. The first aspect involves the amount of information which should be acquired from the client along with the client's request for a target language screening tool. This amount of implementation depends entirely on the implementation of the Validator PDA hosted on the application tier. Our previous description of the PDA-based non-linear versioning system explains the use of Validator PDAs in the context of

Authorized licensed use limited to: UNIVERSITY OF CONNECTICUT. Downloaded on November 16,2020 at 16:26:38 UTC from IEEE Xplore. Restrictions apply.

validating software versions, and hence the current version of software under the client's use is part of the appropriate information which should be collected. However, Validator PDAs are modifiable by nature, in order to accommodate different rules. For instance, software developers or system administrators may want to grant clients access to certain tools, not based on the current version of the tools they are using, but rather a custom priority-based access system. In this case, implementation of the framework must take into consideration proper encoding scheme of user priority level, in such a way that the resulting object can be validated by a PDA. Another development aspect involves selecting a feasible client-server architecture on a per-case basis. While the example provided in Figure 5 shows a three-tier client-server architecture, actual implementation is subjected to available resource and other constraints. As such, other architectures which offer higher ease of maintenance and faster client-server communication speeds may be desirable.

IV. Conclusion

This paper highlights that existing versioning infrastructure of common web-based software systems borrow largely from linear UR models, because the linearity offers simplicity in terms of implementation. This has led to a number of inconveniences down the road, such as the lack of flexibility to create branched versions for common software update and rollback purposes, and the increased storage cost due to duplication of files involved as part of different versions of a particular target software. Based on these inconveniences, a new PDA-based versioning framework has been proposed, and the inner workings of this conceptual framework has been explained, based on two scenarios in which the inconveniences caused by linear versioning systems can be improved. The examples that have been discussed in this paper show that the PDA-based framework for software versioning has solid theoretical foundation and simple conceptual basis, and can be implemented with relative ease. With respect to the proposed language screening central repository server, further research into the implementation feasibility of the PDA-based versioning framework shall proceed in the direction of prototyping a web-based language screening tool as a minimal working example of the framework.

Acknowledgment

We would like to express our deepest appreciations to all staff of both the School of Computer Sciences and the School of Humanities, USM. This study was supported by the short-term grant (304/PHUMANITI/6315349) from Universiti Sains Malaysia. We would also like to extend our gratitude to Dr. Gan Keng Hoon for the aid provided throughout the research process in order to obtain feasible implementations of the PDA-based framework. The results of this paper would not have been possible without a thorough understanding of context-specific applications of pushdown automata, which has been made possible through the material and references provided.

References

[1] Jain, N., Mali, S. G., & Kulkarni, S. (2016). Infield firmware update: Challenges and solutions. 2016 International Conference on Communication and Signal Processing (ICCSP). doi:10.1109/iccsp.2016.7754349

[2] Takagi, T., & Furukawa, Z. (2010). The Pushdown Automaton and Its Coverage Criterion for Testing Undo/Redo Functions of Software. 2010 IEEE/ACIS 9th International Conference on Computer and Information Science. doi:10.1109/icis.2010.144

[3] Chandy, K. M., Browne, J. C., Dissly, C. W., & Uhrig, W. R. (1975). Analytic models for rollback and recovery strategies in data base systems. IEEE Transactions on Software Engineering, SE-1(1), 100–110. doi:10.1109/tse.1975.6312824

[4] Lakhani, H., Tahir, R., Aqil, A., Zaffar, F., Tariq, D., & Gehani, A. (2013). Optimized Rollback and Re-computation. 2013 46th Hawaii International Conference on System Sciences. doi:10.1109/hicss.2013.434

[5] Ge-Ming Chiu, & Cheng-Ru Young. (1996). Efficient rollback-recovery technique in distributed computing systems. IEEE Transactions on Parallel and Distributed Systems, 7(6), 565–577. doi:10.1109/71.506695

[6] Bauwens, J., Ruckebusch, P., Giannoulis, S., Moerman, I., & Poorter, E. D. (2020). Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles. IEEE Communications Magazine, 58(2), 35–41. doi:10.1109/mcom.001.1900125

[7] Chandra, H., Anggadjaja, E., Wijaya, P. S., & Gunawan, E. (2016). Internet of Things: Over-the-Air (OTA) firmware update in Lightweight mesh network protocol for smart urban development. 2016 22nd Asia-Pacific Conference on Communications (APCC). doi:10.1109/apcc.2016.7581459

[8] Baresi, L., Ghezzi, C., Ma, X., & Manna, V. P. L. (2017). Efficient Dynamic Updates of Distributed Components Through Version Consistency. IEEE Transactions on Software Engineering, 43(4), 340–358. doi:10.1109/tse.2016.2592913

[9] J. Schröpfer, F. Schwägerl and B. Westfechtel, Consistency Control for Model Versions in Evolving Model-Driven Software Product Lines, (2019) ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Munich, Germany, 2019, pp. 268-277, doi: 10.1109/MODELS-C.2019.00043.

[10] Jakubec, K., Polák, M., Nečaský, M., & Holubová, I. (2014). Undo/Redo Operations in Complex Environments. Procedia Computer Science, 32, 561–570. doi:10.1016/j.procs.2014.05.461

[11] Waite, M. C., Theodoros, D. G., Russell, T. G., & Cahill, L. M. (2010). Internet-Based Telehealth Assessment of Language Using the CELF–4. Language Speech and Hearing Services in Schools, 41(4), 445. doi:10.1044/0161-1461(2009/08-0131)

[12] Roever, C. (2006). Validation of a web-based test of ESL pragmalinguistics. Language Testing, 23(2), 229–256. doi:10.1191/0265532206lt329oa

[13] Roever, C. (2001). Web-Based Language Testing . Language Learning & Technology, 5(2), 84–94. http://dx.doi.org/10125/25129

[14] Baker J, Kohlhoff J, Onobrakpor SI, Woolfenden S, Smith R, Knebel C, Eapen V. (2020). The Acceptability and Effectiveness of Web-Based Developmental Surveillance Programs: Rapid Review JMIR Mhealth Uhealth 2020;8(4):e16085

[15] Anderson, A. (2014). Web-based Telerehabilitation Assessment of Receptive Language. Washington State University Master Thesis). https://s3.wp.wsu.edu/uploads/sites/867/2015/08/Anderson_2014.pdf

[16] Kelly, J. P. J., Avižienis, A., Ulery, B. T., Swain, B. J., Lyu, R.-T., Tai, A., & Tso, K.-S. (1986). Multi-Version Software Development. IFAC Proceedings Volumes, 19(11), 43–49. doi:10.1016/b978-0-08-034801-8.50013-1

Authorized licensed use limited to: UNIVERSITY OF CONNECTICUT. Downloaded on November 16,2020 at 16:26:38 UTC from IEEE Xplore. Restrictions apply.