The Snippet Platform Architecture – Dynamic and Interactive Compound Documents

Laurent Kirsch, Jean Botev, and Steffen Rothkugel

Abstract—**Compound documents in desktop environments face many issues and limitations in terms of modularity and genericity with regard to their representation and handling. The Snippet Platform natively supports highly dynamic and interactive compound documents. The contents of a document are not limited to combinations of traditional types like text, images, vector objects, or videos, but are completely generic and thus may even consist of fully-functional embedded applications. These can be operated in-place, enabling a higher level of interactivity. Moreover, complex dependencies within or in between documents are supported. The component-based architecture of the Snippet Platform allows for keeping these in a consistent working state. It realizes application software by an extensible set of functional modules that are designed to interoperate. These may also be combined for performing more advanced transformations on contents or defining more refined algorithmic dependencies. A fine-granular model, based on so-called *Snippets* instead of files, ensures the persistent representation of such compound documents and their dependencies. A working prototype, also discussed in this paper, incorporates all these aspects.**

Index Terms—**Data management, document engineering, dynamic compound documents.**

I. Introduction

While creating, editing and viewing offline, stand-alone documents, it is often desirable to integrate static content types with more dynamic, interactive content types similar to websites. These could involve text, images, pictures or videos, but also games, translators and many other kinds of elements. Current, desktop-based systems however do not support the tool interoperation required for enabling such a level of genericity in local document authoring and representation. Traditionally, documents are managed by separate authoring tools and therefore can only consist of contents that the respective, individual tool is able to handle. Some authoring tools allow for extending their standard functionality by means of plugins, but this is not the general case [1], [2] and seriously hampers the authoring process.

Content elements which are automatically adapted as a result of modifications to other elements, are only supported by and within specific applications. For example, a given cell value in a spreadsheet may be the average of other cell values. Inter-document dependencies, if at all, remain mostly unsupported. In general, the functionalities of different authoring tools cannot easily be combined for creating or maintaining documents [1]. In fact, current tools mostly rely on custom, often proprietary data formats, which entails that different tools for authoring the same type of content usually cannot handle each other's documents [2]. As a consequence, tools for authoring content of a given type often provide overlapping functionality with only minor functional differences. New tools cannot simply provide complementary functionalities and rely on the fact that users can edit content by combining various tools. The formats generally differ and the conversion between tools is mostly either unavailable or leading to unsatisfactory results.

Taking a design perspective, state-of-the-art authoring tools are usually monoliths [3], [4] that provide redundant functionality and interoperate poorly. In contrast to that, for instance traditional Unix utilities are well-established and proven in this respect. They are small tools for performing a specific task and can be connected for more complex operations [5], [6]. This is made possible via standard mechanisms – supported by every application-level utility – which are greatly facilitated by the fact that Unix tools and documents are text-based only [7]. Modern authoring systems, however, need to handle many different types of content and combinations.

The *Snippet Platform* natively supports the integration of traditional content such as text, images, videos, or vector objects, and more dynamic, interactive elements like embedded applications. Moreover, it enables dependencies between elements in the same but also different documents. While the Snippet Platform's concepts extend into the UI layer [8], [9], this paper discusses how it natively supports dynamic and interactive content in documents. Section II illustrates an advanced authoring process as enabled by the Snippet Platform. The following section covers the key architectural concepts, which allow for combining the functionality of many authoring tools, as well as the integration of arbitrary content types in the same document. Section IV then explains how these concepts provide means for embedding application functionality into documents, and how they support dependencies. The presented concepts are already implemented in a prototypical emulation of the system, which is the focus of Section V. Finally, related work is reviewed in Section VI, before summarizing and discussing the contributions in Section VII.

II. Advanced Authoring

Consider a professor of computer science creating slides for introducing a new programming language to students. The basic concepts of the programming language are covered on a series of introductory slides, discussing fundamental

Manuscript received August 20, 2013; revised November 1, 2013.

The authors are with the Faculty of Science, Technology and Communication at the University of Luxembourg, 1359 Luxembourg (e-mail: {laurent.kirsch, jean.botev, steffen.rothkugel} @uni.lu).

elements and syntactical features based on various examples. For allowing international students to immediately follow the explanations on the slides, the professor may want to create translated versions of all or certain slides. These do not need to be created from scratch, as dedicated dependencies can be defined for the text boxes containing the explanations. In order to not manually define such dependencies for every text box, the slide show's set of master slides can be configured for automatically creating translated versions. The translation can further be limited to a set of selected text boxes, which allows for including content, such as a list of the programming language's keywords, which needs to remain unchanged. Class hierarchies and other structural information can be automatically converted from the editable code listings on the slides into UML diagrams and vice versa. This allows for picking the currently clearest and most convenient presentation both with regards to the analysis of changes and their implications, but also to the students' individual preferences.

To explain the programming environment itself, it can be embedded right into the slides, allowing for an in-line demonstration of the most common operations, such as the compiling and debugging of programs. For this, not only the editor view with the source code, but also the tool bars and helper views can be embedded by means of a simple preference change. On other slides, only the editor view can be embedded for discussing source code examples. As embedded contents can be manipulated in-place, this allows for easily showing how specific modifications to the source code affect the operation of the compiled program. For a more convenient demonstration, the standard output of the program can also be included in the slides. To avoid recompiling and rerunning the program after every single modification, a custom compile-and-run dependency can be defined between the content of the editor view and that of the output view. Subsequent updates to the source code then trigger an update of the displayed output without any explicit action from the professor. Moreover, the view displaying the output also shows the compiler's debug messages in case of incorrect modifications so that the students better understand and identify the implications of the various error types.

In principle, any dependency which can be described by means of an algorithmic operation may be defined between entire documents or individual parts. Moreover, any types of traditional content and applications can be integrated in the same document. As will be detailed in the following, this is made possible by the Snippet Platform's realization of application software by functional modules and the communication mechanisms these support.

III. Architecture

The Snippet Platform's component-based architecture guarantees that authoring tools and other applications are realized by functional modules, so-called *Typed Shells* (*T-Shells*) and *Typed Commands* (*T-Commands*). These operate on a well-defined content type and constitute the Snippet Platform's GUI-based counterparts to Unix shells and commands. This allows for a maximum level of genericity in terms of their coupling, i.e., the integration and interoperation with respect to different functionality. The architecture guarantees that T-Shells and T-Commands support all communication mechanisms required for jointly managing documents and also keeping dependencies between individual elements in a consistent state. A T-Shell provides an environment for creating, modifying and displaying content of a specific type. Its graphical user interface consists of the canvas into which it renders the managed content, and the UI elements, e.g., context menus and toolbars, which allow the user to manipulate that content. It triggers the operations corresponding to individual menu entries or toolbar buttons on demand. These operations are realized as T-Commands, each of which is a separate component that implements specific functionality for producing or modifying content of a given type. Functionality that can be invoked through the provided UI controls thus depends on the available T-Commands. As all components are typed, T-Shells are able to identify and retrieve compatible T-Commands and to adapt their UI elements as new functionality is added and removed again. In fact, T-Commands are classified according to several functional criteria which enable T-Shells to group similar functionality, thus allowing for a better user experience.

The architecture also includes a fine-grained document model that persistently represents compound documents – consisting of various content types potentially authored by different components – through graphs of interrelated *Snippets* [10] that constitute specific content elements. Thus, the contents maintained by different components can be combined in the same document, while remaining separate Snippets. This enables each involved component to define which content elements are represented by individual Snippets and to interlink them for optimizing the resulting graphs. Moreover, Snippets can be shared among individual graphs, a mechanism leveraged for representing dependencies between content elements of any type and granularity which can be part of different documents. While constituting an additional conceptional layer on top of documents, dependencies reuse the same architectural mechanisms at the level of persistent representations, namely Snippet graphs.

In the following, T-Shells, T-Commands, and how they interoperate for jointly managing documents, will be discussed in more detail. Moreover, the persistent representation of compound documents by Snippets will be introduced. The mapping of more interactive, dynamic content onto these architectural mechanisms is then described in Section IV.

A. T-Shells

The Snippet Platform's architecture provides the mechanisms required for combining content managed different T-Shells in the same document. T-Shells, each handling content of a specific type, can be nested and combined into tree-based hierarchies that reflect the relationships between the individual parts of a document. Thus, arbitrary contents can be integrated and authored in an entirely flexible fashion. Every T-Shell supports a set of pre-defined communication interfaces, which enable the embedding of external content, i.e., content it does not handle

by itself, into own content and vice versa. Figure 1 shows two concrete examples. Document (a) is a leaflet intended for printing with some text and a UML class diagram, while Document (b) is a slide show involving source code and the output of the corresponding compiled utility. As illustrated to the right in the figure, every distinct document part is managed by its own instance of a T-Shell, even when multiple parts consist of the same content type. For instance, in case of the slide show, both the source code and the standard output are texts that are managed by different instances of a text-handling T-Shell. This approach simplifies the internal architecture of T-Shells as it reduces the communication management overhead.

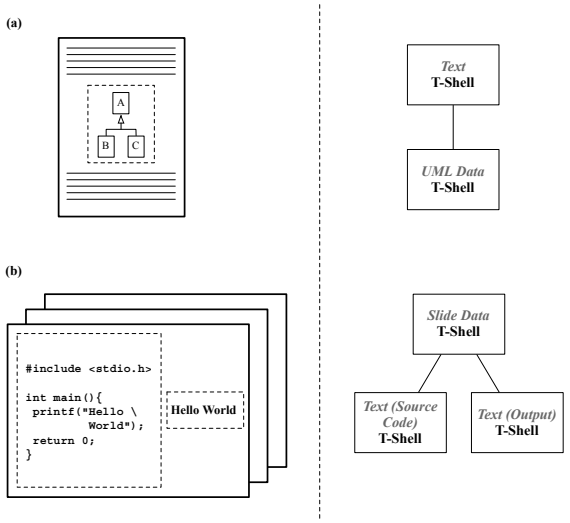

Fig. 1. Two Documents and the T-Shells managing them.

The communication interfaces between T-Shells allow for handling layout-related issues. For instance, when the canvas containing the UML diagram of Document (a) is resized, the managing T-Shell, in addition to scaling the diagram, also notifies the T-Shell which handles the enclosing text. The latter then reflows its text around the embedded canvas' new frame. In certain scenarios, such notifications also need to be sent after content updates. For instance, as a result of a preference change, the text-handling T-Shell of Document (a) may have to flow its text around the UML diagram instead of its frame. When elements at the outside are resized, repositioned, deleted, or when newly added elements become the outermost objects, the T-Shell needs to be notified.

B. T-Commands

T-Commands are separate components that realize operations on content of a given type. These operations can create, modify, or supply additional information. T-Commands can be invoked by and connected to any other T-Command and any T-Shell, provided that the components involved operate on the same content type. Pre-defined T-Shells and types exist for ubiquitous content such as rich and plain text, images, videos, or vector objects. This guarantees that T-Commands operating on such content are based on the same type definitions and can be connected for more complex operations. Moreover, a standard set of T-Commands, which provide common functionality, is an integral part of the Snippet Platform. Thus, redundant implementations of simple, frequently-required functionality are effectively avoided.

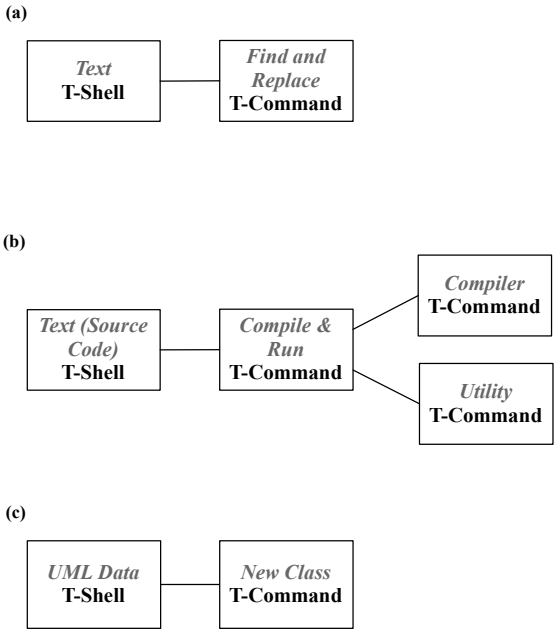

Fig. 2. Example T-Commands

Fig. 2 shows some concrete examples of T-Commands. In Scenario (a) existing content is modified by a T-Shell that manages text, which invokes a T-Command for performing a find-and-replace operation. T-Commands may, however, also provide additional information on content, as is illustrated in Fig. 2(b). Here, a run-and-compile script realized as a T-Command is invoked by a T-Shell on the source code that it handles for obtaining the output of the matching compiled utility. For this, the script invokes two other T-Commands, the first of which compiles the source code and the second of which constitutes the utility just compiled. The latter is run when compilation was successful.

For enabling their invocation by the user and by other T-Commands, these can be parametrized in different ways. For instance, a find-and-replace T-Command needs to know the string to be replaced and its substitution string. Depending on the parameterization, it either operates on the arguments provided at its invocation, or it supplies a view that allows the user to enter the two required strings. Moreover, the results of a T-Command's operation can be used in different ways. Depending on the parameters provided at invocation, T-Command (b) which compiles and runs a utility may present the final output in a dedicated view, return it to the invoking component for further processing or for adding it as a comment to the source code itself. T-Commands can also create new content, as shown in Fig. 2(c), where the tools for adding elements to UML diagrams are implemented as T-Commands. Here, a T-Command for adding a new class to a UML class diagram has been invoked.

C. Snippets

Every document is represented by a graph of Snippets, each constituting a well-defined content element. This induces a finer granularity of data and allows for matching the structure of persistent representations to that of the actual content. Snippet graphs are created and maintained by the T-Shells that manage the corresponding content elements. As

illustrated in Fig. 3, distinct parts of a document, which are managed by different T-Shells, are separated into distinct subgraphs. This enables every T-Shell to define which content elements are represented by individual Snippets. The Snippets of each subgraph are organized as an ordered tree, with a special Snippet at its root, which stores information on the maintaining T-Shell.

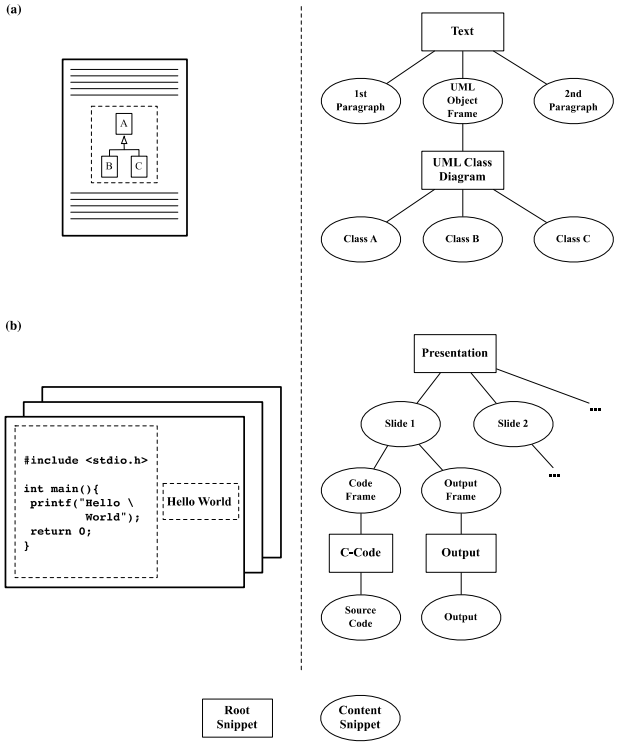

Fig. 3. Two documents and their snippet graphs

With regards to the matching of structure with content, in case of the UML class diagram every distinct class for instance is represented by a separate Snippet, while in case of the slide show, individual slides are represented by distinct Snippets, each of which links the Snippets representing the elements of the corresponding slide. The ordered subgraph can reflect the sequence of the represented elements, such as paragraphs of text or a slide deck. An inherent advantage of reflecting document structure at the level of Snippet graphs is that parts which are likely to be accessed as a whole can be retrieved more easily. Here, for instance, the data of a specific slide or class only can be accessed or modified conveniently. Moreover, the structure of the resulting graph can be leveraged for performing some of the most common operations. For example, moving elements between slides or reordering the slides is equivalent to updating links between Snippets.

IV. Modelling Dynamic, Interactive Content

The pursued architectural approach allows for novel ways of authoring dynamic and interactive documents. Most importantly, it allows for integrating also generic application functionality with traditional contents such as text, images, or videos. This integration is discussed in Section IV-A. Moreover, dependencies between parts of the same or different documents are supported. Section IV-B then describes dependencies and how they are kept in a consistent state.

A. Embedding Application Functionality

In the Snippet Platform, all software offering the functionality of traditional applications such as media players, internet browsers or games, is composed of the same architectural components, T-Shells and T-Commands. The design of a T-Shell enables it to use its canvas not just for displaying conventional contents, but also elements of user interfaces which are fully operational.

The slide show in Fig. 4 for instance includes a code editing view managed by the T-Shell handling the source code. Realizing operations on content as T-Commands here allows for debugging the source code without ever leaving the context of the slide. When invoked by the T-Shell, the T-Command that constitutes the debugger provides the required controls, all or some of which can be embedded within the code editing view itself. According to the user's preferences, controls then are shown either in the view with the source or in helper views hidden from the audience in full-screen mode. The communication interface between the T-Shell and the debugger enables the T-Shell to highlight the current instruction in the code listing. Moreover, the debugger may request the T-Shell to add a specific debugging state to the Snippet graph of the source code. This permits resuming the debug process later on, or freezing a given state, for instance, to discuss it with other people.

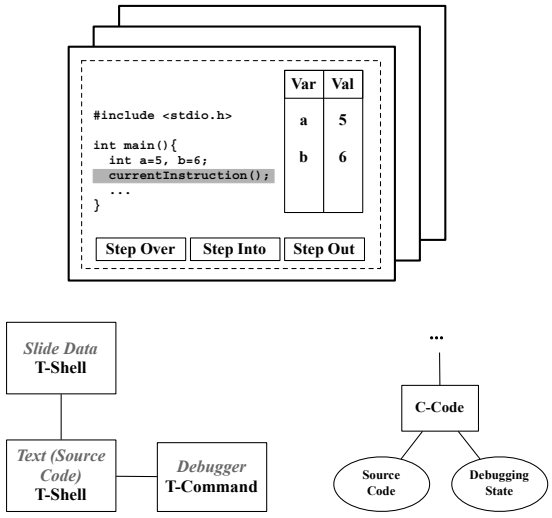

Fig. 4. Slide show with embedded code editing view and embedded debugging controls

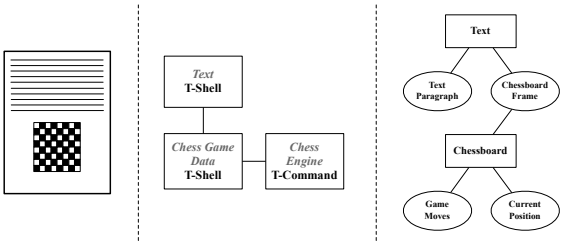

Fig. 5. Text document with embedded game interface.

Fig. 5 showcases another example involving an embedded game. A text document on a famous chess game includes an interactive chessboard for playing through the game right within the document. The board is managed by a dedicated T-Shell handling the notations of chess games and the

corresponding Snippet graphs. Such a graph consists of a game's moves and information on the position currently visible on the board. Like any T-Shell, the one handling the board may also invoke T-Commands. Here for instance, a chess engine is invoked which analyzes the position and evaluates different move alternatives. The communication interface between the engine and the T-Shell allows, for instance, the highlighting of the best move on the board.

The generic architecture thus allows for easily embedding any kind of functionality into documents by the same set of mechanisms.

B. Dependencies

A dependency is a relation based on an algorithmic operation which is performed on a set of document parts as input. This operation results in another dependent document part which then is automatically adapted in case of modifications to the input, guaranteeing that the dependency remains in a consistent state. For this, the operation is rerun on the modified input parts and the contents of the dependent document part are updated. The operation itself is realized by a T-Command and therefore depends on the set of available commands in its expressiveness. Dependencies may also be chained, i.e., the dependent document part in a given dependency may constitute an input part to another dependency. Cycles, however, are prevented for avoiding infinite update loops.

Fig. 6 shows an example for such a chained dependency sequence containing source code, the standard output of the compiled utility, and a Chinese translation of the output. It is ensured that following modifications, the source code is recompiled and the compiled utility is run afterwards for updating the output. A translator is then invoked on the new output for adapting the Chinese translation.

Despite constituting an additional conceptional layer on top of documents, dependencies and documents are based on the same architectural mechanisms. Dependencies are also represented as graphs of Snippets while at the same time the finer granularity allows for defining dependencies between arbitrary content elements. Whether the document parts involved are elements of the same or different documents does not affect the Snippet graph of a dependency. It merely consists of links to the graphs that constitute the individual parts. Thus, the corresponding Snippets are shared among the dependency graph and the original document graph. For conciseness, only the root Snippets of the graphs are depicted. Moreover, the graph of a dependency links a document with information on the T-Command that implements the operation which creates the dependent document part from the input parts. In the given example, that is a simple script which recompiles and reruns the utility in case of source code modifications and a translator which subsequently updates the Chinese translation of the output.

An update of a document part within a dependency can be either triggered by a modification of one of its input parts, or by a modification of the part itself. If, in the latter case, the part also constitutes the dependent part, the dependency turns invalid and its graph is deleted. Upon any modification of input parts the dependency is kept in a consistent state by updating the dependent document part. For example, when altering the source code, the managing T-Shell notifies the system as illustrated in Fig. 7. The system then inspects the graphs of existing dependencies and subsequently invokes the T-Commands which update them. In case of the source code, there is only one dependency that is updated by the script recompiling the source code, then running the utility just compiled for obtaining the new output. The output is returned to the system, which asks the managing T-Shell to replace the previous output. Finally, that T-Shell notifies the system, which then invokes the translator on the output just generated for adapting the Chinese translation.

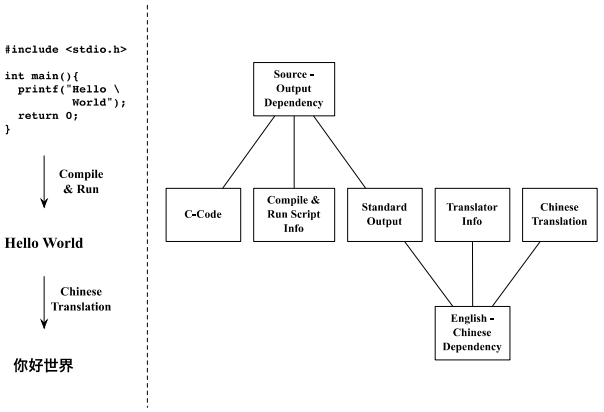

Fig. 6. Document parts with dependencies and the representing snippet graphs.

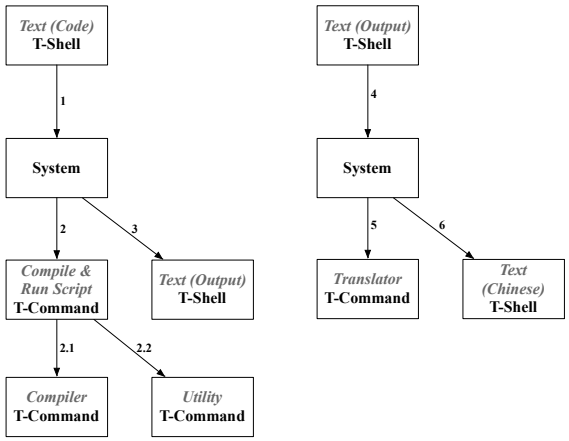

Fig. 7. Dependency updates.

V. Prototype

An application-level emulation of the Snippet Platform has been developed. It serves as a proof-of-concept and realizes all aspects described thus far, i.e., Snippets, T-Shells, T-Commands, the embedding of application functionality, and dependencies. The Snippet model itself has been implemented on top of an object database, which provides many desirable mechanisms out of the box, e.g., the simplified persistency of a Snippet graph.

The current prototype also comprises a framework for creating T-Shells, T-Commands and the associated type definitions. A set of modules and abstract classes facilitate the development of custom types and functionality. They guarantee the presence of required mechanisms in all T-Shells and T-Commands, thus also fostering a coherent user experience across the entire Snippet Platform. It is, for instance, ensured that a dependency between content elements can be created in the same fashion, independently of the managing T-Shells. For testing purposes, an extensible set of T-Shells and T-Commands – along with the associated

type definitions – has been developed. These operate on text, vector objects, slide shows, images, videos and mind maps. Moreover, there is a calculator and a function plotter, which are realized as T-Shells and T-Commands operating on numbers respectively function equations. As detailed in Section IV-A, this realization guarantees that their fully-operable interfaces can be integrated with traditional content. Finally, for verifying and showcasing the practicability and usefulness of dependencies, the system component which allows for keeping these in a consistent working state has also been emulated.

VI. Related Work

In the following, approaches and methodologies related to the Snippet Platform are described. Section VI-A discusses existing frameworks that enable the interoperation of authoring tools and support compound documents, while Section VI-B presents alternatives to Snippets for persistently representing such documents.

A. Compound Document Frameworks

While the Snippet Platform constitutes an entire system architecture, existing approaches are mostly realized as an add-on to established platforms, which limits their capabilities in terms of enabling the interoperation of application software and supporting interactive, dynamic content. In fact, these approaches normally focus on a subset of the Snippet Platform's concepts. For instance, many existing compound document frameworks like Object Linking and Embedding (OLE) [11], OpenDoc [12], KParts [13], Open Object Embedding [14], HotDoc [15], or Fresco [16], target the pure nesting of documents, user interfaces, or both. Microsoft's OLE, for example, focuses on the linking and embedding of documents and individual content elements. However, it does not enable tools that can be connected for complex tasks. KParts, a component framework for the KDE desktop environment, focuses on making specific application functionality available to other applications via so-called parts. These are mostly elaborate, GUI-based widgets. For instance, the Konqueror file and web browser, for providing file previews, relies on viewer widgets delivered with the fully-fledged applications. However, connectable, non-GUI-based utilities for processing data are not supported. Generally, their realization as an add-on to established platforms entails that the approaches mentioned before cannot support dependencies between content elements of different documents. In fact, the underlying platforms do not support the required inter-tool communication.

HTML and Javascript also enable the integration of static with dynamic and interactive content. However, a compound document system based on these technologies requires a set of additional layers for interacting with the hardware as well as interpreting and rendering the HTML and Javascript. The Snippet Platform here adopts a holistic approach with its native layers already being optimized for the handling of compound documents consisting of multiple, possibly interactive and dynamic content types, from the representation and storage layer over the fully-flexible and extensible functional layer to the user interface itself. This way, compared to a non-native approach both a better performance and a better user experience can be guaranteed [17]. Moreover, Snippets can constitute or contain any kind of data, including HTML and Javascript themselves. This enables the developer of a new T-Shell to choose the most appropriate data format for representing the content type involved.

B. Compound Document Representations

Many of the approaches discussed above use custom file formats for storing compound documents. A major drawback here is that these formats are generally very complex. For instance, an entity perceived as a single file by the user may actually consist of a small file system containing many subfiles [3], [18]. However, unlike a native graph as enabled by Snippets, these file systems within a file do not offer the same flexibility in terms of matching the structure of the persistent representation to that of the actual content. Moreover, sharing subfiles between many files is not possible, which seriously compromises the definition of dependencies across document boundaries.

Another common and standardized approach for storing compound documents is XML. Indeed, XML namespaces enable the integration of arbitrary contents, potentially created by many different tools. Technologies such as XLink and XPointer allow for representing dependencies between arbitrary document parts. However, unlike Snippets which are designed for maximum native genericity in terms of document dynamics and interactivity, XML introduces an intermediary layer, which comes at the cost of lower efficiency, for instance when parsing document representations [19]. It should also be noted that while the Snippet model itself is not XML-based, XML and Snippets can still be used jointly. T-Shells for instance can store XML data within Snippets, if appropriate for the supported content type.

Object databases and object-oriented database management systems like db4o or the Zope Object Database (ZODB) constitute a further option for storing compound documents. They provide features such as automatic indexing with generic retrieval options. The Snippet Platform also comprises these features, but through its key set of extensible abstractions which are utilized in all implementation layers, it offers a fully integrated and even thus more specialized approach.

VII. Conclusion

This paper discusses the design and implementation of the *Snippet Platform*, a compound document system whose concepts natively allow for a maximum degree of modularity and genericity in document authoring. The content of a document is not limited to traditional elements such as plain and rich text, images, videos, or vector objects, but also the integration of arbitrary embedded application functionality is supported. The automatic adaptation of content elements following modifications to other elements is another key feature. Furthermore, algorithmic dependencies can be defined between document parts of any granularity both within and across individual document boundaries. The component-based architecture realizes application software

by a set of functional modules that are able to interoperate. Their functionality can also be combined, thus enabling the definition of more complex dependencies. Snippets as a fine-granular document model support these dependencies and the combination of many content types in the same document with a persistent graph representation. They can be shared among multiple graphs, which enable the persistent representation of dependencies. Moreover, Snippet graphs allow for combining contents authored and maintained by different components in the same document, while keeping them separated into different subgraphs. As a result, every component may organize the subgraphs that it manages for performing some of the most common operations by updating the graph structure itself.

The Snippet Platform's key concepts, most importantly the realization of all application software as interoperating functional modules and the fine-granular representation of documents by means of Snippets, constitute an important step towards dynamic and interactive documents in traditional desktop environments.

REFERENCES

[1] M. Grechanik, D. S. Batory, and D. E. Perry, "Integrating and reusing GUI-Driven applications," in *Proc. the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, Springer-Verlag, 2002, pp. 1–16.

[2] M. Grechanik and K. M. Conroy, "Composing integrated systems using GUI-Based applications and web services," in *Proc. the IEEE International Conference on Services Computing*, IEEE, 2007, pp. 68–75.

[3] T. Harter, C. Dragga, M. Vaughn, A. C. A. Dusseau, and R. H. A. Dusseau, "A file is not a file: Understanding the I/O behavior of apple desktop applications," *ACM Transactions on Computer Systems*, vol. 30, no. 3, pp. 10:1–10:39, 2012.

[4] B. Lampson, *Computer systems research – past and present. SOSP 17 keynote lecture*, 1999.

[5] M. Gancarz, *The UNIX Philosophy*, Digital Press, 1995.

[6] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Communications of the ACM*, vol. 17, no. 7, pp. 365–375, 1974.

[7] E. S. Raymond. (2003). The Art of UNIX Programming. [Online]. Available: http://www.faqs.org/docs/artu/

[8] L. Kirsch, M. Esch, and S. Rothkugel, "The snippet system – fine-granular management of documents and their relationships," in *Proc. the 6th IASTED International Conference on Human- Computer Interaction*, ACTA Press, 2011.

[9] L. Kirsch, J. Botev, and S. Rothkugel, "An extensible tool set for creating and connecting reusable learning resources," in *Proc. World Conference on Educational Media, Hypermedia and Telecommunications 2012. EdMedia 2012.* AACE, 2012, pp. 1434–1442.

[10] L. Kirsch, J. Botev, and S. Rothkugel, "The Snippet system – reusing and connecting documents," in *Proc. the Seventh International Conference on Digital Information Management. ICDIM 2012.* IEEE, 2012, pp. 138 – 144.

[11] K. Brockschmidt, *Inside OLE*, Microsoft Press, 1995.

[12] Apple Inc., "OpenDoc - Shaping tomorrow's software," 1993, white Paper.

[13] L. Bölöni, *Programming KDE 2.0: Creating Linux Desktop Applications*, Publishers Group West, 2000.

[14] B. E. Backlund, "OOE: A compound document framework," *SIGCHI Bulletin*, vol. 29, no. 1, pp. 68–75, 1997.

[15] J. Buchner, "HotDoc: A framework for compound documents," *ACM Computer Survey*, vol. 32, no. 1es, 2000.

[16] M. Linton and C. Price, "Building distributed user interfaces with fresco," in *Proc. the Seventh X Technical Conference*, 1993, pp. 77–87.

[17] N. P. Huy and D. V. Thanh, "Evaluation of mobile app paradigms," in *Proc. the 10th International Conference on Advances in Mobile Computing & Multimedia*. ACM, 2012, pp. 25–30.

[18] D. Rentz, "OpenOffice.org's documentation of the microsoft compound document file format," *Tech. Rep.*, 2003.

[19] T. Lam, J. Ding, and J. C. Liu, "XML document parsing: operational and performance characteristics," *Computer*, vol. 41, no. 9, pp. 30 –37, 2008.

Laurent Kirsch received his master degree in computer science from the University of Luxembourg in 2010. Since November 2010 he is a doctoral candidate and teaching assistant at the University of Luxembourg. His research focuses on document engineering, interactive systems and personal information management.

Jean Botev is currently a postdoctoral researcher at the computer science and communications research unit of the University of Luxembourg. He received his diploma degree in computer science and media studies from the University of Trier (Germany) in 2007, followed by a PhD in Computer Science from the University of Luxembourg in 2011. His research interests include complex networks, self-organization and collaborative socio-technical systems.

Steffen Rothkugel received a Ph.D. in computer science from the University of Trier, Germany, in 2001. Since 2002 he is associate professor in the computer science and communications research unit of the University of Luxembourg, where he is heading a small team. His work and teaching revolve primarily around the domains of system software and distributed systems. His current research focuses on interactive distributed systems and document engineering.