HUSKY: A Spreadsheet for End-User Service Composition

Daniel Skrobo
Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
daniel.skrobo@fer.hr

Abstract

End users need a simple and intuitive methodology for service composition. HUSKY is an end-user environment designed for composing services based on spreadsheet paradigm. This paper presents a spreadsheet-based paradigm which incorporates the concept of time that enables end-users to intuitively express concurrency through a visual arrangement of activities within spreadsheet cells.

1. Introduction

In this paper a novel end-user methodology for service composition development is proposed. This methodology enables users to combine services by making service compositions in a way that no application developer could foresee. For example, instead of just ordering a book from an on-line store, most of us would like to buy the book at a minimum price. Figure 1 shows an example of how existing services can be organized into a process for finding the best price quote for a list of books. After deciding which books to buy, the user may use on-line bookstore services from FooBooks.com and BarBooks.com to get the price quotes for the books. After comparing the offers, a decision is made on who will fulfill the book orders.

Figure 1. The process for finding a best price quote for books

The technological basis for development of services is Web Service Resource Framework (WSRF) [1] that defines a set of platform-independent standards for building services. In addition, services developed using heterogeneous technologies can be composed into coherent service compositions using standard XML-based languages like WS-BPEL (Web Services Business Process Execution Language) [2]. However, due to their complexity, existing programming languages and development tools are not appropriate for end-users without formal programming skills. Instead being focusing on development of application, users are burdened with formal concepts and rules of the language.

The environments for which end-users build their composite services are inherently distributed, concurrent, and nondeterministic. Parallel and distributed programming is a demanding task even for professional software engineers [3]. However, given the appropriate methodology, end-users will be able to cope with non-determinism and organize available services into meaningful applications. The world we live in is nondeterministic and we always find ways to reduce this non determinism at certain points in time and space to produce valid results. For instance, in book pricing example, the user knows that finding price quotes from independent booksellers could be done concurrently. Thus, he or she could assign price quote tasks to his friends and afterwards share information to make a final decision.

2. Service Composition Methodologies

To find the most suitable methodology that lowers the cognitive burden [4] for end-user application development, existing models for service composition have been evaluated, ranging from text-based languages [5] to wizard-based [6] approaches. Although commonly used methodologies for service composition are based on graphical dataflow/wiring diagrams [7], they are not taken into account because modeling of complex applications is demanding for screen representation.

To evaluate suitability of text-based and wizard-based methodologies for end-user modeling, the past experiences from the development of the PIE (Programmable Internet Environment) framework are used. The PIE framework is a service-oriented middleware for distributed system development based on CL [8] and SSCL [9] languages. The CL (Coopetition Language) language is an XML business process description language based on extensions to standard WS-BPEL. The SSCL (Simple Service Composition Language) is a script-based coordination language with lightweight syntax rules. To make the distributed system
development less error-prone, the PIE framework supports wizard-based construction of SSCL programs.

Since CL and SSCL languages are primarily designed for distributed system development, they provide a high level of control and expressiveness inappropriate for end-user application modeling. CL and SSCL languages contain concepts that end-users are usually not familiar with, such as variables and their values [10]. In addition, these languages are sequential so they do not enable end-users to define concurrent processes intuitively.

The wizard-based extension of SSCL language, however, is strongly structured in nature, and thus inflexible for end-user needs. The aim is to allow users to assemble composite services without a burden of predetermined development practice.

3. HUSKY: Spreadsheet Service Composition Methodology

To bring development closer to end-users, a new methodology named HUSKY\(^1\) (HUman-centered Service composition workspacE and methodology) has been designed. HUSKY is a methodology based on spreadsheets. The development of HUSKY is based on two assumptions. First, the representation of service composition can be organized in a two-dimensional space that resembles the process of sketching an idea with a pencil on a sheet of paper. Second, of all the graphical paradigms for programming languages, spreadsheets are widely accepted by end-users. Cells of a spreadsheet are used as basic programming elements with familiar properties that are adapted for building service compositions. Spreadsheet-like environments are most suitable for end-users, since the majority of users find it easier to perform calculations in spreadsheets than by writing an equivalent sequential program. Furthermore, the spreadsheet workspace panel enables visual organization of task activities in a way similar to graphical schemes, yet easily readable on the screen.

3.1. Service composition

Service composition consists of a set of service invocations. For example, if two services need to execute sequentially, this is modeled by putting service invocation events into two adjacent cells. Service invocation events in HUSKY are denoted by the Execute keyword. Cells [A2-A3] in Figure 2 show an example of service invocations. To invoke a service, the service location and the operation name must be specified. The service location is specified in a separate cell to improve readability of service invocation events. For example, cell [A2] contains the event for invocation of the Input service from Figure 1:

Execute [C2] “GetInput”

where Execute determines that the cell contains a service invocation event, [C2] specifies a cell in which the service location is defined, while “GetInput” specifies the operation that the invoked service should perform. Note, the keywords in the HUSKY spreadsheets are typed in bold.

Cells [C2-C3] contain service definitions for Input and FooBooks services. For example, cell [C2] contains the definition of the Input service:

Service “http://www.input.hr”

where Service determines that the cell contains a service definition, while “http://www.input.hr” is the endpoint URL where the service is available.

After execution of the Execute event in cell [A2], this cell will contain the result of the execution of the Input service. The result of the Execute event in cell [A2] is used as input parameter for the execution of the Execute event in cell [A3]:

Execute [C3] “GetPrice” [A2]

where [A2] specifies the cell that contains input data for GetPrice operation of the FooBooks service.

3.2. Time ordering

Service compositions require events in the cells to be correlated in time. Hence, the first important requirement in designing the HUSKY environment is to empower users with an intuitive way of expressing the timing in service composition. Time ordering relation is introduced, which transforms the spatial organization of events into their ordering in time. The time ordering relation within a HUSKY workspace is defined along two dimensions: horizontal and vertical. Time progresses from left to right and from top to bottom. Two events are sequential in time if they occupy two adjacent cells in a HUSKY workspace. A set of adjacent cells makes a sequence of

\(^1\) Just as husky dogs are harnessed into a team to drag a sled, so are the services combined into service composition to provide new functionalities.
events, while empty cells disjoin the workspace into temporally independent event sequences.

The example of service composition shown in Figure 3 consists of three temporally independent sequences of events: A, B, and C. Sequence A, defined in cells [A2-A5], consists of events \( A = \{a_1, a_2, a_3, a_4\} \). Another two sequences \( B = \{b_1, b_2, b_3, b_4, b_5\} \) and \( C = \{c_1, c_2, c_3, c_4\} \) are given in cells [C1, D1-D4] and [A7-B7, B8-C8], respectively. The arrows denote the time ordering relation within sequences.

Upon application start, all event sequences start concurrently executing events in their left most and upper most cells. Since event sequences are temporally independent, the execution of each sequence proceeds within its local time which is measured by a local clock. Ticks of the clock are aligned with the cells that comprise the sequence. The local clock makes a tick each time an event of the sequence is executed. For example, local clock of sequence A in Figure 3 makes the following ticks: \( A_2, A_3, A_4, \) and \( A_5 \). The local sequence clock is controlled using two types of events. For example, if cell [A5] contains the event

\[
\text{Set Clock to [A2]}
\]

the local clock of sequence A will be set to A2 and the execution of the sequence will continue from the event in cell [A2]. This would cause the sequence of events in cells [A2-A5] to repeat indefinitely.

Another type of event changes the local time only if certain condition is met. For example, if the cell [D1] contains the event

\[
\text{If [C1] Then Set Clock to [D3]}
\]

the local clock of sequence D will be set to D3 if the value stored in cell [C1] is true. This will skip the execution of event in cell D2 and start the execution of event in cell D3. However, if the value in cell [C1] is false, local clock of sequence B advances automatically to D2 causing the execution of event in cell D2.

4. Service collaboration

Each sequence of events is temporally independent from any other sequence within a HUSKY workspace. Separating the sequences of events enables the user to cope with the non determinism inherent in service composition, which is necessary to model activities that are inherently distributed and concurrent. Although event sequences are temporally independent and self contained, sometimes it is necessary to define interactions among them. Inter sequence interactions introduce the minimal determinism necessary to produce valid results in a given service composition. Thus, the second important requirement in the design of the HUSKY environment is to enable modeling of dependencies between sequences of events. Temporally independent sequences of events may need to exchange data through communication, synchronize their activities in time, and collaborate through publish-subscribe interaction model. To model these interactions, the HUSKY environment introduces four types of special-purpose objects: Clipboard, Queue, TokenCenter, and BrokerCenter.

Figure 4 presents a HUSKY implementation of the book-best price quote application shown in Figure 1. In this example, the usage of Clipboard, Queue, and TokenCenter is presented. Cells [A1-A5] include definitions of services, while cells [C2-C8], [G2, H2-H3], and [G5-G7] define three event sequences that coordinate and control service composition execution. The sequence in cells [C2-C8] invokes the Input service to retrieve a list of books specified by the user, Decision service to select book-best offer, and Output service to inform the user about the best offer. The event sequences [G2, H2-H3] and [G5-G7] invoke the BarBooks and FooBooks services, respectively, with the list of books received from user. Cells [E2], [E4], and [E8] include definitions of TokenCenter, Clipboard, and Queue objects, respectively. Sequences use Clipboard to exchange list of books and Queue to exchange book quotes. TokenCenter is used to postpone the invocation of FooBooks and BarBooks services until results from the Input service are acquired.
4.1. Communication

Communication between sequences of events is enabled through Clipboard and Queue objects. Clipboard is a storage object that provides well-known copy-and-paste mechanism used in the majority of end-user applications. A Clipboard can hold one value at a time, while its contents may be retrieved many times. When new value is copied into a Clipboard, the old value is overwritten. A Clipboard is useful when value produced by one event is intended for multiple use by other events. Figure 4 shows an example of using a Clipboard for inter-sequence communication. Cell [E4] contains a definition of Clipboard object used by the following event in cell [C4]:

**Copy [C2] to [E4]**

where Copy specifies an event for copying a value from cell [C2] to cell [E4]. The sequences defined in cells [G2, H2-H3] and [G5-G7] use the value stored in Clipboard [E4] for execution of the Execute events in cells [H2] and [G6]. Events in cells [H2] and [G6] use value stored in the Clipboard to invoke the operations "GetQuote" and "GetPrice" of the services defined in cells [A2] and [A1], respectively.

The Queue object is a communication service that holds multiple values with a restriction that a particular value may be retrieved only once. When a new value is put into a Queue, it is added to the set of values already stored in the Queue. When a value is acquired from a Queue, the value is removed from the Queue and is no longer available. Figure 4 shows an example of using a Queue where the event sequences [G2, H2-H3] and [G5-G7] send values to the event sequence [C2-C8]. The communication consists of two events: a Put event to add a value to the Queue and a Get event to acquire a stored value from the Queue. The Queue object is defined in the cell [E8]. For example, cell [H3] contains a Put event:

**Put [H2] to [E8]**

where Put and to determine an event for adding a value to a Queue. The parameter [H2] specifies the cell that contains data, while the parameter [E8] specifies the cell in which the Queue is defined.

Cells [C5] and [C6] contain a Get event:

**Get [E8]**

where Get to determine an event for acquiring a stored value from a Queue.
Get [E8]

where Get determines an event for acquiring a value from a Queue. The parameter [E8] specifies the cell in which the Queue is defined. After execution, cells [C5] and [C6] will contain the result of the Get event.

4.2. Synchronization

The TokenCenter object enables mutual exclusion and synchronization of events in two or more concurrent event sequences. Figure 4 presents an example of synchronization of concurrent event sequences defined in cells [C2-C8], [G2, H2-H3], and [G5-G7]. In this example, the event in cell [C2] must run before the events in cells [H2] and [G6]. A TokenCenter is defined in cell [E2] to synchronize these sequences of events:

**TokenCenter Initially "0"

where the TokenCenter represents the definition of a TokenCenter object, while Initially defines the initial number of tokens.

Synchronization consists of two events: Get Token and Return Token, which acquire tokens from and return them to the TokenCenter.

Cells [G2] and [G5] contain a Get Token event:

**Get Token [E2]

where Get Token determines an event for acquiring a
token from the TokenCenter. The parameter [E2] specifies the cell in which the TokenCenter is defined. If a token is not available in TokenCenter, execution of the sequence will be suspended until at least one token becomes available through execution of the Return Token event elsewhere in the HUSKY spreadsheet.

Cell [C3] contains a Return Token event:

**Return Token [E2] Count “2”

where Return Token determines an event for returning a token to TokenCenter. The parameter [E2] specifies the cell in which the TokenCenter is defined, while Count “2” specifies the number of tokens to be returned to the TokenCenter.

4.3. Publish-subscribe cooperation

The BrokerCenter object enables content-based message delivery by using a publish-subscribe cooperation model. In addition to publishers and subscribers, brokers are the third party that cooperates through BrokerCenter. Publishers announce information to the BrokerCenter. Subscribers register brokers and terms of interest to the BrokerCenter. Broker analyzes information announced by publishers according to the subscribed terms. If the announced information satisfies the terms, the broker triggers the subscriber by sending a notification. Terms, announcements, and notifications are

![Figure 5](image_url)

**Figure 5. Implementation of the book-best price quote application in HUSKY by using Clipboard, Queue, and TokenCenter objects**
Figure 5 presents the use of BrokerCenter for implementation of the book-best price quote application shown in Figure 1. In this example, the terms specify the list of books for which the subscriber wishes to find the lowest price. Publishers send announcements containing book pricing information. If the book is listed in the subscriber’s terms, the broker sends a notification with the name of the book seller with the lowest price.

Event sequences defined in cells [A3-A4] and [C3-C4] publish information to a BrokerCenter defined in cell [A6]. The event sequence defined in cells [C8-C10] subscribes to the BrokerCenter. The Broker service defined in cell [E6] analyzes announcements from publishers and triggers the subscriber.

The BrokerCenter object is defined in cell [A6]. An example of a Broker service definition is given in cell [E6]:

**Broker** "http://www.brokers.com/BestPrice"

The Broker defines a Broker object, while the literal “http://www.brokers.com/BestPrice” represents the URL at which the Broker service is available.

The cell [A4] presents an example of a Publish event definition:

**Publish BrokerCenter** [A6] **Announcement** [A3]

The parameter [A6] specifies the cell that contains the definition for BrokerCenter, while the parameter [A3] specifies the cell that contains the announcement that will be published.

The cell [C9] defines an example of Subscribe event:

**Subscribe BrokerCenter** [A6] **Broker** [E6] **Terms** [C8]

The parameter [A6] specifies the cell that contains the definition for BrokerCenter, the parameter [E6] specifies the cell that contains the definition of Broker service, while the parameter [C8] specifies the cell that contains the terms of subscription.

5. Spreadsheet framework for service composition

Service compositions developed using HUSKY are based on a hierarchical service composition model, while execution of composite services is supported by a distributed execution environment. HUSKY service composition framework is presented in Figure 6.

5.1. Hierarchical service composition model

HUSKY service compositions are based on a hierarchical service composition model that brings together end-users, application developers, and service developers in a unified development process. End-users use HUSKY Editor to express the ordering of events in time within a service composition in form of a spreadsheet. Since HUSKY users are not required to understand the complex service composition technologies, they can focus on effectively expressing the core logic of their service compositions.

Application developers are responsible for specifying all the details of service composition that were not explicitly specified in HUSKY spreadsheet. To that end, HUSKY spreadsheets are translated into distributed programs implemented in a high-level script language such as Python, or a service composition language such as WS-BPEL. Application developers augment logic of distributed programs with additional logic, like conversion of service parameter data formats, which is required for correct execution of service compositions.

Service developers are service composition technology experts that understand the application-specific logic of the given service composition. They are responsible for implementation of custom services used by given service composition and final tuning of the distributed program logic. Service developers use compilers for translating WS-BPEL distributed programs into programs written in a high-level programming language like Java, C++, or C#. Alternately, they may also use a linker tools to augment the Python distributed programs with logic written in Java, C++, or C#.

5.2. HUSKY editor

The HUSKY Editor allows users to write inline comments of arbitrary text in-between keywords that specify definitions and events. The comments unobtrusively complement the keywords that make service compositions more readable. For example, the sequence [G2, H2-H3] of the book-best price quote application presented in Figure 4 may include the following inline comments

**Get** Token from token center at [E2]

**Execute** service BarBooks at [A2] and perform operation “GetQuote” with book list [E4]

**Put** book price [H2] obtained from BarBooks service to queue [E8]

5.3. Execution environment

Service execution environment enables translation and execution of service compositions written in HUSKY spreadsheets. As shown in Figure 6, the execution environment consists of a set of interpreters, collaboration services, and custom services.

Distributed programs are scheduled and executed on a set of machines that host interpreters for high-level programming languages. The functionalities of the Clipboard, Queue, TokenCenter, and BrokerCenter are implemented through special-purpose collaboration services. The Collaboration libraries run on top of
interpreters and provide distributed programs with support for invocation of collaboration services. Collaboration services and libraries are provided out-of-the-box and serve for collaboration of distributed programs. Finally, a service composition designed in a HUSKY spreadsheet is realized by execution of distributed programs, which in turn invoke custom services and interact through the collaboration services.

For example, the sequence \([G2, H2-H3]\) of the book-best price quote application presented in Figure 4 is translated to a distributed program with the following Python code:

```python
G2 = TokenCenterLib.GetToken("http://www.fer.hr/TokenCenter/E2"
)
E4 = ClipboardLib.Paste("http://www.fer.hr/Clipboard/E4"
)
)
H3 = QueueLib.Put("http://www.fer.hr/Queue/E8", H2
)
```

The distributed program is formed primarily using Python calls of methods in collaboration libraries that use XML for data representation and communication. The TokenCenterLib, ClipboardLib, QueueLib, and ServiceLib are collaboration libraries for TokenCenter, Clipboard, Queue, and custom service invocation, respectively.

### 6. Summary

Spreadsheets have matured and evolved from the pioneering efforts of Mattessich [11], and Pardo and Landau [12] in the sixties, and widely adopted systems like VisiCalc [13] in the seventies, and Lotus 1-2-3 and Excel [14] in the eighties and nineties. Each new generation of these tools adopted technological advances of its period. VisiCalc used personal computers to introduce the concept to large audiences; Lotus pushed the complexity and interoperability envelope on PC hardware; while Excel created new experiences leveraging a full GUI environment, and after much evolution, turned itself towards the Internet through its web query capability. Looking at more recent efforts, on line spreadsheets like Google spreadsheets will focus on leveraging the Web and an interconnected computing infrastructure that will increase communication and cooperation and further improve access anytime from anywhere.

HUSKY is the first service composition system leveraging spreadsheets, thus pioneering an effort to weave Internet services with an intuitive paradigm that has proven its advantages over the years. The presented research results targets two key areas. First, the spreadsheet paradigm has been extended by adding a concept of time. Second, intuitive expression of concurrency and non determinism of service composition is accomplished through a visual arrangement of service activities within spreadsheet cells.

The spreadsheet is an intuitive interface that led to its widespread adoption, with the number of spreadsheet users far outnumbering skilled software developers. HUSKY methodology works to reduce the gap between these two groups. The experiences from the past have shown that to solve a given service composition problem, it takes a couple of hours in HUSKY, days in script based languages like SSCL, and weeks in XML-based languages like WS-BPEL.

### 7. References


