Proceedings of the of the 17th Conference on Computer Science and Intelligence Systems pp. 913–917
DOI: 10.15439/2022F188
ISSN 2300-5963 ACSIS, Vol. 30

An Experimentation Framework for Specification and Verification of Web Services

Szymon Katra
Warsaw University of Technology
Faculty of Electronics
and Information Technology
Nowowiejska Str. 15/19
00-665 Warsaw, Poland
Email: szymon.katra.stud@pw.edu.pl

Wiktor B. Daszczuk
Warsaw University of Technology
Institute of Computer Science
Nowowiejska Str. 15/19
00-665 Warsaw, Poland
Email: wiktor.daszczuk@pw.edu.pl

Denny B. Czejdo
Fayetteville State University
Department of Mathematics and
Computer Science
Fayetteville, NC 28301, USA
Email: bczejdo@uncfsu.edu

***Abstract*—Designing and implementing Web Services constitutes a large and constantly growing part of the information technology market. Web Services have specific scenarios in which distributed processes and network resources are used. This aspect of services requires integration with the model checkers. This article presents the experimentation framework in which services can be specified and then formally analyzed for deadlock-freedom, achievement of process goals, and similar features. Rybu4WS language enriches the basic *Rybu* language with the ability to use variables in processes, service calls between servers, new structural instructions, and other constructions known to programmers while remaining in line with declarative, mathematical *IMDS* formalism. Additionally, the development environment allows simulation of a counterexample or a witness - obtained as a result of the model checking - in a similar way to traditional debuggers.**

I. Introduction

A RISING number of available Web Services are used for business processes in the modern world. Interactions with other Web Services are key features in creating more complex scenarios and satisfying business needs. An example of interaction between different services might be a travel agency that uses external services to book hotels, flights, and other facilities. Service for booking flights may use another service for processing a payment that communicates with a bank or credit card provider. Such interaction between many services is called *Web Service composition* [1]. From the computer science point of view, Web Service composition is a distributed system concerned with typical problems like deadlocks or lack of termination..

In Warsaw University of Technology, Institute of Computer Science, an experimentation framework for specification and verification of web services composition was developed. It is based on IMDS formalism (Integrated Model of Distributed Systems [2]) and DedAn [3] tool to model asynchronous distributed systems and verify them automatically. The user does not need to have deep knowledge about verification methods such as temporal logic and model checking.

A distributed system under verification in IMDS formalism is defined as a set of actions used to model the behavior. The declarative input of the DedAn environment was designed to structure a set of actions by combining them across servers or agents. While DedAn input language is sufficient to specify simple distributed systems, modeling more complex cases is challenging due to various technical difficulties described later in this paper.

To overcome the problem of modeling complex distributed systems, a higher-level language *Rybu* was initially developed [4], which simplifies the modeling of the system by some imperative-style elements and data aggregation. To improve the modeling of Web Service composition, *Rybu4WS* language was created, which is the original contribution of this paper. Moreover, Rybu4WS Debugger tool was developed to visualize counterexamples or witnesses caught from DedAn directly on the original Rybu4WS code. The latter feature is unique among model checking tools: they verify the systems but do not allow to interpret the checking results on the source code of the tested system. The projection of the verification result onto the source form of the specification is one of the most significant achievements of the authors.

This paper is organized as follows: Section 2 covers related work of web service composition. Architecture of the Experimentation Framework is in section 3. Section 4 gives a brief description of IMDS formalism and DedAn tool. Description of Rybu4WS and its syntax can be found in section 5. General conversion rules of Rybu4WS code to IMDS model are described in section 6. Section 7 contains a description of the Rybu4WS Debugger tool. Conclusions and possible future development of our experimentation framework are covered in section 8.

II. Related Work

Labeled Transition Systems (LTS) are alternative approaches for modeling Web Service compositions [5] where transitions between states can represent Web Service interactions. In the mentioned paper, model-checking and temporal logic properties were used to verify the Web Service composition modeled using this approach.

Existing formalisms like CSP [6] or CCS [7] are well designed to model concurrent systems, but they are hardly suitable for distributed systems. They do not possess asynchronous features needed for modeling true distributed systems. Instead, they rely on synchronous communication in the system, which requires that communicating processes reach given states simultaneously before passing a message. Such a scenario is impossible in Web Services or any other distributed system because components are typically placed on separate machines in different locations. They cannot learn about the other party's state in other ways than by message exchange. However, there were attempts of formal software verification based on Service Component Architecture (SCA) [8] converted into CSP specification [9].

Bandera [10] tool allows the creation of a finite-state transition model directly from Java source code that can be verified in the external model checker. The main goal of this project was to provide automated model extraction from software systems that allows easy verification without manual software analysis and model creation. While automated creation of a model from the source code could be very convenient, generated abstraction might affect the model precision. As an alternative to verification, automated WS testing is proposed. Combinatorial method is described in [11] and metamorphic in [12]. Fault injection testing is presented in [13]. Simulation is covered in [14].

There are also languages specifically designed for writing distributed programs, like SR language (Synchronized Resources) [15] that provide various mechanisms used for concurrent process interaction. However, it lacks the ability of formal verification and is not suitable for model checking.

Widely used in industry WSDL [16] format describes Web Service interfaces for other services or applications. Since WSDL is designed to specify the pure interface of Web Services, it is not possible to define the internal behavior of Web Service, which is necessary for verification against deadlocks or checking termination.

A significant number of studies were conducted about Web Service composition, for example hybrid approach [14]. Report [15] presents different automatic composition approaches. TripICS [16] is an example of a real-life application that uses automatic WS composition for planning trips and travels around the world. It is based on the PlanICS framework to solve automatic composition problems, which uses a combination of SMT-solver and genetic algorithms [16]. Automated WS composition for Financial Decision Support is presented in [17]. Semantic modeling is covered in [21][22].

III. Architecture of the Experimentation Framework for Web Service Composition

To provide efficient experimentation with Web Service Composition, a modular but highly integrated system was created. Rybu4WS program is converted to IMDS form and checked by DedAn verifier, then the witness/counterexample is caught by the Rybu4WS Debugger which can simulate the verification output directly on the source Rybu4WS code.

When DedAn is run with user interface, additional analysis facilities become available, like export to Uppaal for checking huge systems, graphical simulation over system components [18], counterexample animation, and detailed analysis of individual components' behavior.

IV. IMDS and DedAn

IMDS [2] formalism is the key element of the experimentation system. Therefore, it will be discussed first. It is a model of a distributed system using that is constructed over a set of actions. The actions are executed in the environment of servers offering services and traveling agents representing distributed computations. The agents use messages for their traveling between servers where partial computations are performed as the execution of actions. A set of messages in given sequential distributed computation forms an agent. The current configuration of a system is defined as a set of states of all servers and a set of current messages of all agents (one message per agent).

Action is a relation between the input pair (message, state) and output pair (new message, new state). The server in a given state accepts the message, which invokes the action. Action execution changes the state of the server and issues a new message. There is a special case on agent termination, which changes only a state without sending a message. The system in IMDS starts from the initial configuration, which consists of initial states for each server and initial messages for each agent.

Formally, the IMDS action is a quadruple of input items and output items ((*message, state*)→(*next message, next state*)) or a triple ((*message, state*)→(*next state*)) (agent terminates).

The IMDS formalism can well represent Web Service composition due to the following essential features:

- Locality - there is no global or non-local state in the system, all servers are independent.
- Autonomy of decisions - server autonomously determines the order of message acceptance.
- Asynchrony of actions – always a message waits for an appropriate state or a state waits for matching message.
- Asynchrony of communication - messages are sent through a unidirectional asynchronous channel.

In most cases, the states of servers and the messages of agents can be treated as atomic, and actions are defined as a relation in $(M\times S)\times(M\times S)$ which defines input message, input state, output message, and output state of an action. Agent-terminating actions are defined in $(M\times S)\times(S)$. The action extracts the input pair (*message, state*) from the input configuration and inserts the pair (*new message, new state*) into the output configuration. The execution of actions is assumed to be in interleaving semantics [2].

The IMDS models can be verified using the DedAn environment, which allows to find deadlocks or check possible termination in the modeled system. The input of DedAn was designed to structure a set of actions by combining them across servers or agents. It allows defining server and agent types used to instantiate variables of those types along with linking them using formal and actual parameters.

V. Rybu for Web Services (Rybu4ws)

It should be emphasized that the set of actions of Rybu/Rubu4WS specification is exactly the same as the set of actions in IMDS specification after conversion. The main role of higher level language is to ease the programming. The instructions in Rybu/Rybu4WS group sets of actions into more readable high level actions, and chain the actions as in imperative language programming.

A Rybu [4] system consists of two kinds of servers: reactive servers and threads (processing servers from which the agents originate). The agent starts its run in a thread and invokes services offered by the servers by means of messages. Invoked service executes an action on the server, changing its state. A server replies from the executed action by sending a message back to the thread, prolonging its execution. The Rybu4WS was developed for modeling Web Service compositions, overcoming the limitations imposed by Rybu. It features more advanced functionality, such as:

- server-server communication that allows agents to travel between different servers and execute complex scenarios,
- state variables in grouped processes, which enables the communication between different processes without sending actual IMDS messages,
- termination at any point of execution,
- complex code sequences in reactive server actions instead of trivial state mutation and return value.

Like Rybu, the Rybu4WS system consists of reactive servers and processes (in Rybu: thread). The reactive server is a resource that holds a particular state and offers services. Each service can be guarded by a condition over variables and contains a code sequence for further actions. Process consists of a code sequence that the agent executes to invoke services on reactive servers and does not hold any state.

Rybu4WS introduces a third, more advanced feature called group, which is used to group one or more processes. It gives the possibility to declare shared variables, allowing the creation of more sophisticated scenarios where processes use the same variables within a server to cooperate.

The following listing presents example Web Service composition in Rybu4WS. It consists of services necessary to build a simple book shop service – warehouse, payment, bank. Processes are used to represent the user's behavior.

```
type BOOL = { t, f };
server Payment {
  var s: { none, pending, paid };
  { Init | s == :none } -> { return :ok; }
  { Confirm | s == :pending } -> {
    s = :paid; return :ok;
  }
  { IsPaid | s == :paid } -> { return :t; }
}
server Bank(p: Payment) {
  var bal: 0..5;
  var s: BOOL;
  { Transfer | bal > 0 && s == :f } -> {
    s = :t; return :confReq;
  }
  { Transfer | bal == 0 || s == :t } -> {
    return :fail;
  }
  { Confirm | s == :t && bal > 0 } -> {
    bal -= 1; s = :f; p.Confirm(); return :ok;
  }
}
server Warehouse() {
  var x: BOOL;
  { Reserve | x == :f } -> { x = :t; return :ok; }
  { Reserve|x == :f } -> { return :outOfStock; }
  { Dispatch | x == :t } -> { x = :f; return :ok; }
}
server BookShop(w: Warehouse, p: Payment) {
  { Begin } -> {
    match w.Reserve() {
      :outOfStock -> { return :fail; }
      :ok -> { p.Init(); return :payReq; }
    }
  }
  { End } -> {
    match p.IsPaid() {
      :t -> { w.Dispatch(); return :ok; }
    }
  }
}
var p = Payment() { s = :none };
var b = Bank(p) { bal = 3, s = :f };
var w = Warehouse() { x = :f };
var bs = BookShop(w, p);
group BookPurchaseScenario {
  var action: { idle, none, pay } = :idle;
  process UserWebInterface {
    match bs.Begin() {
      :fail -> { action = :none; terminate; }
      :payReq -> {
        match b.Transfer() {
          :confReq -> {
            action = :pay; bs.End(); terminate;
          }
          :fail -> { terminate; }
        }
      }
    }
  }
  process UserMobileApp {
    wait(action != :idle);
    if (action == :pay) { b.Confirm(); }
    terminate;
  }
}
```

A. Reactive servers

The reactive server in Rybu4WS consists of variables, actions, and dependencies.

Variables form the internal state of the server, which can be used in conditions for action and can be mutated by the actions code.

An action defines the behavior of a service. It includes an optional predicate, a condition over state variables used to determine whether an agent can execute the given action in the current server state or not, and a code sequence for execution. The code sequence might contain service calls to other servers, variable mutations, return statements, process termination, or conditional statements. Each service has a unique name used by other servers or processes for calling the service. In case a server state satisfies the predicate of more than one action in a given service, the action to execute is chosen non-deterministically. The code sequence is a sequence of statements executed when an action is invoked.

The collection of other servers needed by the given server is called dependencies. Only servers defined in the dependency list can be called from the server.

In order to use reactive servers, an actual server instance must be created. Initial state and required dependencies must be defined for each reactive server instance. This allows creation of many servers with the same behavior but with different initial states or dependencies.

B. Processes

A process is a code sequence with an accompanying agent. It is used as an entry point for agent execution. Each process is converted into one IMDS server and a single IMDS agent. Ungrouped processes (group will be explained later) can only call instantiated reactive servers in the system and cannot define any variables.

C. Groups

A group is a collection of processes and variables. The group's primary goal is to enable processes to use shared variables for sophisticated business scenarios where processes can communicate without sending actual IMDS messages. Agents are instantiated like in ungrouped processes, one agent per one process, meaning that many agents can work simultaneously on the same variables.

VI. Conversion to IMDS

Rybu4WS language is used only for modeling distributed systems and cannot be directly verified against deadlocks or terminations. For the purpose of verification in the DedAn environment, Rybu4WS code must be converted into IMDS equivalent using a set of unambiguous translation rules. More detailed description of Rybu4WS language and architecture of the environment is available in [19].

It is important to note, that during conversion, the original code locations of each statement are preserved in IMDS states and messages. In a later stage, they are used to visually present deadlock or termination/non-termination scenarios directly on the Rybu4WS code after verification in DedAn.

Each Rybu4WS reactive server instance is converted to a state machine. Every state machine represents a single IMDS server.

Server variables are converted to IMDS states exactly like in Rybu - they are defined by a Cartesian product of sets of all possible values of the variables in the server.

The process is converted into a state machine and corresponding agent instance. In comparison to Rybu4WS reactive servers, it does not provide any services that could be externally invoked. The process consists of a single code sequence block that is converted into a state machine and provides a special service used by agent as an entry point.

Group is converted into a single, encompassing state machine by merging state machines created for each process. Additionally, the group can contain variables that are converted as in the reactive servers. Many agents can run concurrently in the same group and share variable values without sending any IMDS message. Each action in the encompassing state machine also includes an agent for which this transition is valid, which means that the agent can travel only within his corresponding code sequence.

VII. Rybu4ws Debugger

The Rybu4WS Debugger [20] is a desktop application that allows Rybu4WS code to be loaded and converted it into a corresponding IMDS representation for the purpose of verification in the DedAn environment. DedAn is invoked automatically (or manually if advanced analysis options are needed) and finds deadlocks/checks termination automatically. It is worth emphasizing that partial deadlocks are identified as well. During verification, a counterexample/witness is elaborated, and visualized in a user-friendly way for the manual analysis. This reverse mapping of the sequence of action onto the source code is achieved because actions in Rybu4WS program are expressed in a more abstract and easily readable form than in IMDS specification. However, the sets of actions in Rybu4WS and IMDS are exactly the same, and the semantics of both specifications is equal.

VIII. Conclusions

The growing number of designed Web Services requires more and more assistance in the programmer's activities, including verification of whether the designed services behave safely (free from deadlocks), whether they finish in inevitable success (process termination), or whether there is even a possibility of success. Tools based on temporal logic are used to verify such behavior.

The formalism used to describe services should be well-suited to distributed systems: support asynchrony, locality of actions, and autonomy of nodes. Ideally, the specification language can be explicitly used for formal verification. Such a modeling method is IMDS, for which the DedAn verification environment has been built. However, the DedAn input language does not fully meet the requirements of programmers; therefore the Rybu language was created and its successor – Rybu4WS. Conveniently for the programmer, it combines the basic IMDS paradigm, adding the possibility of coupling actions in reactive servers. It was achieved using a syntax close to the programmers' habits, using typical control statements such as conditional branching, loops, and response handlers. Shared variables in process groups allow to easily communicate by mutation of variable values.

The severe limitation of Rybu – allowing the server to be called only from a process – was solved: now a call chain can be created. Additionally, it is possible to terminate the process without returning it to the home server Communication between servers, declaring shared variables for processes, and termination at any point of execution gives the ability to model Web Service compositions

The Rybu4WS Debugger tool provides a user-friendly interface that allows analyzing counterexamples similarly to debuggers in usual programming environments. It is a backward engineering principle, seldom observed in typical verifiers: they produce counterexamples or witnesses that are not easy to analyze in the context of the source code.

Rybu4WS can be used to model a wider variety of concurrent distributed systems than just Web Service themes. The development needs of such systems would require additional programming elements, which could eventually lead to the creation of a Domain Specific Language (DSL) family that might be the subject of further research. It is currently impossible to create a circular dependency between reactive servers, meaning that callbacks or recursive calls are not supported. Also, it is not possible for a single agent to "split" and perform multicast action, i.e., calling multiple services in a parallel manner. That would allow the creation of agents traveling in the distributed system and not returning to the place they originate from.

REFERENCES

[1] B. AL-Shargabi, A. El Sheikh, and A. Sabri, "Web Service Composition Survey: State of the Art Review," *Recent Patents Comput. Sci.*, vol. 3, no. 2, pp. 91–107, Jun. 2010. doi:10.2174/2213275911003020091

[2] W. B. Daszczuk, "Specification and Verification in Integrated Model of Distributed Systems (IMDS)," *MDPI Comput.*, vol. 7, no. 4, pp. 1–26, Dec. 2018. doi:10.3390/computers7040065

[3] W. B. Daszczuk, "Using the Dedan Program," in *Integrated Model of Distributed Systems*, Cham, Switzerland: Springer Nature, 2020, pp. 87–97. doi: 10.1007/978-3-030-12835-7_6

[4] W. B. Daszczuk, M. Bielecki, and J. Michalski, "Rybu: Imperative-style Preprocessor for Verification of Distributed Systems in the Dedan Environment," in *KKIO'17 – Software Engineering Conference, Rzeszów, Poland, 14-16 Sept. 2017*, 2017, pp. 135–150. https://arxiv.org/abs/1710.02722

[5] M. Ghannoudi and W. Chainbi, "Formal verification for Web service composition: A model-checking approach," in *2015 International Symposium on Networks, Computers and Communications (ISNCC), Yasmine Hammamet, Tunisia, 13-15 May 2015*, 2015, pp. 1–6. doi: 10.1109/ISNCC.2015.7238576

[6] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978. doi:10.1145/359576.359585

[7] R. Milner, *A Calculus of Communicating Systems, LNCS vol. 92*, vol. 92. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980. ISBN 978-3-540-10235-9

[8] H. Paik, A. L. Lemos, M. C. Barukh, B. Benatallah, and A. Natarajan, "Service Component Architecture (SCA)," in *Web Service Implementation and Composition Techniques*, Cham: Springer International Publishing, 2017, pp. 203–250. doi: 10.1007/978-3-319-55542-3_8

[9] W. Chargui, T. S. Rouis, M. Kmimech, M. T. Bhiri, L. Sliman, and B. Raddaoui, "Towards a formal verification approach for service component architecture," in *SOMET 2017: 16th International Conference on Intelligent Software Methodologies, Tools, and Techniques, Kitakyushu, Japan, 26-28 Sept 2017*, 2017, pp. 466–479. doi: 10.3233/978-1-61499-800-6-466

[10] J. C. Corbett, M. B. Dwyer, and J. Hatcliff, "Bandera: a source-level interface for model checking Java programs," in *22nd international conference on Software engineering - ICSE '00, Limerick, Ireland, 4-11 June 2000*, 2000, pp. 762–765. doi: 10.1145/337180.337625

[11] I. Bluemke, M. Kurek, and M. Purwin, "Tool for Automatic Testing of Web Services," in *5th International Workshop Automating Test Case Design, Selection and Evaluation, FEDCSIS, Warsaw, Poland, 7–10 Sept 2014*, 2014, pp. 1553–1558. doi: 10.15439/2014F93

[12] I. Bluemke and A. Sawicki, "Tool for Mutation Testing of Web Services," in *13th DEPCOS/Reclomex, Brunów, Poland, 2-6 July 2018*, 2019, pp. 46–55. doi: 10.1007/978-3-319-91446-6_5

[13] S. Ilieva, I. Manova, and D. Petrova-Antonova, "Towards a methodology for testing of business processes," in *7th Federated Conference on Computer Science and Information Systems, FEDCSIS, Wroclaw, Poland, 09-12 Sept 2012*, 2012, pp. 1315–1322. https://ieeexplore.ieee.org/document/6354354

[14] T. Preisler, T. Dethlefs, and W. Renz, "Simulation as a Service: A Design Approach for large-scale Energy Network Simulations," in *10th Federated Conference on Computer Science and Information Systems, FedCSIS, Lodz, Poland, 13-16 Sept 2015*, 2015, pp. 1765–1772. doi: 10.15439/2015F116

[15] G. R. Andrews *et al.*, "An overview of the SR language and implementation," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 1, pp. 51–86, Jan. 1988. doi:10.1145/42192.42324

[16] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL)," 2001. https://www.w3.org/TR/wsdl.html

[17] L. Belava, "Concept of Platform for Hybrid Composition, Grounding and Execution of Web Services," in *11th Conference on Advanced Information Technologies for Management, FEDCSIS, Kraków, Poland, 8–11 Sept 2013*, 2013, pp. 1071 – 1077. https://annals-csis.org/Volume_1/pliks/190.pdf

[18] G. Baryannis and D. Plexousakis, "Automated Web Service Composition: State of the Art and Research Challenges," 2010. https://publications.ics.forth.gr/tech-reports/2010/2010.TR409_Automated_Web_Service_Composition.pdf

[19] A. Niewiadomski, P. Switalski, M. Kowalczyk, and W. Penczek, "TripICS - a Web Service Composition System for Planning Trips and Travels," *Fundam. Informaticae*, vol. 157, no. 4, pp. 403–425, Jan. 2018. doi:10.3233/FI-2018-1635

[20] I. Pawełoszek, "Integrating Semantic Web Services into Financial Decision Support Process," in *11th Conference on Advanced Information Technologies for Management, FEDCSIS, Gdansk, Poland, 11-14 Sept 2016*, 2016, pp. 1189–1198. doi: 10.15439/2016F99

[21] S. De, P. Barnaghi, M. Bauer, and S. Meissner, "Service modelling for the Internet of Things," in *6th Federated Conference on Computer Science and Information Systems, FedCSIS, Szczecin, Poland, 18-21 Sept 2011*, 2011, pp. 949–955. https://ieeexplore.ieee.org/document/6078180

[22] S. Demirkol, M. Challenger, S. Getir, T. Kosar, G. Kardas, and M. Mernik, "SEA_L: A Domain-specific Language for Semantic Web enabled Multi-agent Systems," in *7th Federated Conference on Computer Science and Information Systems, FEDCSIS, Wroclaw, Poland, 09-12 Sept 2012*, 2012, pp. 1373–1380. https://ieeexplore.ieee.org/document/6354358

[23] W. B. Daszczuk, "Graphic modeling in Distributed Autonomous and Asynchronous Automata (DA^3)," *Softw. Syst. Model.*, vol. 20, no. 5, pp. 363–398, 2021. doi:10.1007/s10270-021-00917-7

[24] S. Katra, "Specification and verification of Web Service composition in DedAn environment," MSc thesis, Dept. of Electronics and Information Technology, Warsaw University of Technology, 2022.