Integrating External Services in DIME

Hafiz Ahmad Awais Chaudhary[1,2(✉)] and Tiziana Margaria[1,2,3]

[1] CSIS, University of Limerick, Limerick, Ireland
ahmad.chaudhary@ul.ie
[2] Confirm - Centre for Smart Manufacturing, Limerick, Ireland
[3] Lero - The Science Foundation Ireland Research Centre for Software, Limerick, Ireland

Abstract. We show how to extend the (application) Domain Specific Languages supported by the DIME low-code development environment to integrate functionalities hosted on heterogeneous technologies and platforms. Developers can this way utilize within DIME entire platforms like e.g. R for data analytics, and collections of services, like e.g. any REST-based microservices. In this paper we describe the current architecture of the DIME-based Digital Thread platform we are building in a collection of interdisciplinary collaborative projects, discuss the role of various DSLs in the platform, and provide a step by step tutorial for the integration of external platforms and external services in DIME. The goal is to enable a wide range of DIME adopters to integrate their application specific external services in the DIME open source platform, bootstrapping a collaborative ecosystem where the low-code activity of integrating external capabilities facilitates an increasingly no-code application development on the basis of pre-integrated Application DSLs.

Keywords: Domain Specific Language (DSL) · Model Driven Development (MDD) · eXtreme Model Driven Development (XMDD) · Service Independent Building Blocks (SIBs) · Low code development environments · DIME

1 Introduction

We address the problem of integrating external services into the DIME integrated modelling environment. As shown in [14], we are building in Limerick a significant ecosystem of collaborations spanning various application domains, with the collective goal of contributing application domain specific collections of services to a DIME-centered low-code application development environment able to use all those services in the application design, in a possibly seamless way. In the terminology that is currently emerging in the advanced manufacturing context, this end-to-end integration of all what is needed to collaboratively deliver a complex, interoperable capability in a potentially cyberphysical cooperation is called the "Digital Thread".

Its definition is not yet settled, but to give an idea, according to the CEO of iBASEt, a company offering Digital Thread products and consultancy, *"the Digital Thread encompasses model data, product structure data, metadata, effectual*

© Springer Nature Switzerland AG 2021
T. Margaria and B. Steffen (Eds.): ISoLA 2021, LNCS 13036, pp. 41–54, 2021.
https://doi.org/10.1007/978-3-030-89159-6_3

data process definition data – including supporting equipment and tools" [6], and *"Product Lifecycle Management (PLM) provides "the what" (modeling, BOM management, process planning, process simulation, and engineering change management). Enterprise Resource Planning (ERP) provides "the when, where, and how much" (scheduling, financials, and inventory). To have a fully developed model-based enterprise—and a fully functioning Digital Thread—manufacturers also need "the how". That's what Product Lifecycle Execution (PLE) provides through process execution, process control, quality assurance, traceability, and deviation handling."*

In our view, the Digital Thread requires an end-to-end integration of the data and processes that guide and deliver such complex operations, and it is our goal to provide this integration and orchestration in a model driven and low-code, formal methods-supported fashion. In this paper, we address specifically the task of integrating external services provenient from various platforms and made available in various programming languages, into the DIME integrated modelling environment. DIME's extension and integration with external systems through the mechanism of native services DSLs extends the capabilities of the core platform to meet wider communication needs (e.g., in the cloud), and also to take advantage of existing sophisticated enterprise services (e.g. AWS).

Low-code programming both at the API and the platform level is considered to be a game changer for the economy of application development. Gartner Inc., for example, predicts [5] that the size of the low-code development tools market will increase by nearly 30% year on year from 2020 to 2021, reaching a $5.8 billion value in 2021. They state that so far, this is the fastest and probably the simplest and most economical method of developing applications.

In this paper, we briefly recall the DIME-based architecture of the Digital Thread platform in Sect. 2, including the central role of Low-code and DSLs in it. Section 3 presents the integration methodology in DIME with a quick tutorial, followed in Sect. 4 details the step by step integration of the R platform and of RESTful services. Finally, Sect. 5 contains our summary and a brief discussion of the perspectives.

2 The Digital Thread Platform in DIME

DIME [3] is a graphical Integrated Modelling Environments for low-code/no-code application development, used to develop research [9,16] as well as industrial applications. It is a general purpose MDD platform-level tool, suitable for agile development due to its rapid prototyping for web application development. It follows the One Thing Approach based on XMDD [18], in a lineage of development environments that traces back to the METAFrame'95 [25,26]. DIME supports both control flow and data flow modelling in its process diagrams. Control flow models admit a single start node but may have multiple end nodes, and nodes (called SIBs) representing single functionalities or sub-models are graphs, i.e. formal models. The SIBs are connected via directed edges depending on the business logic, with distinct edge types for dataflow and control-flow.

Software systems in general, and especially web apps in internet-centered ecosystems and digital threads in an Industry 4.0 context, are not isolated in nature: they demand interaction with various external systems, libraries and services. Frequent needs are (but not limited to)

- acquire sensors data from external systems,
- feed data to external dashboards for analytics and publishing,
- utilize the compute power of cloud systems,
- reuse sophisticated enterprise services.

We will use DSLs to virtualize the technological heterogeneity of the services, delivering a simple, coherent and efficient extension to this low-code modelling platform. The extension by integration adds to the tools the capability to communicate with sophisticated enterprise ecosystems, without sacrificing the flexible yet intuitive modelling style for the no-code users, who just use the DSLs that are available.

2.1 The Current Architecture

The current architecture of the Digital Thread platform is depicted in Fig. 1 (from [14]).

DIME's own **Language DSL** (in orange), designed in Cinco [21], encompasses for the moment in our application settings primarily the Data, Process and GUI models. This is the layer defining the expressive power of the DIME modelling language. Data, Process and GUI models are used to design the applications, thus the Digital Thread platform makes use of these facilities "as is". We do not extend nor modify this layer.

The concrete applications designed and implemented in DIME (in blue) use the modelling and orchestration language of the Language DSL, and as vocabulary a number of service collections provided in directly in the core DIME platform (the Common DSLs, e.g. concerning the supported GUI elements and their functionalities), but also a growing collection of Process DSLs that may be application specific or still generic.

All these are part of the **Application DSL** layer, that includes also quite a number of **Native DSLs** external to DIME (in green). These DSLs collect the atomic SIBs that expose to the application design layer the encapsulated code provenient from a rich and growing collection of external service providers. For example, in [8] and [7] we have integrated a number of EdgeX Foundry services [1] that support a variety of IoT devices and relative communication protocols. The same has happened for AWS and other services in [4].

The DSLs may concern specific application domains, but the parameter that determines the specific kind of integration is the technology on which they run. To give and example, one can use data analytics in R to analyze cancer-related data, census data, proactive maintenance data in the manufacturing domain, or risk and insurance related data in financial analytics. As long as they use the same R functions, the integration in DIME is exactly the same and needs to be

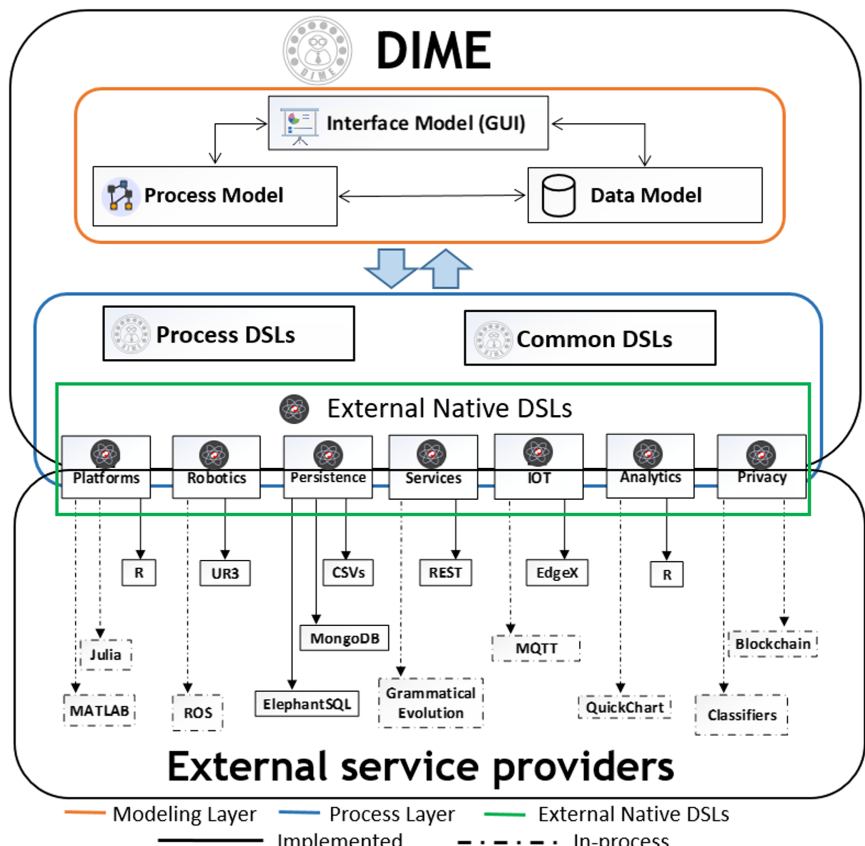

Fig. 1. Architecture Overview of DIME and Custom DSLs

done just once. The same applies for example to Julia and Matlab: we will address the integration of a DSL for each of them in the context of a biomechanical simulation collaborative project, but the corresponding DSLs will be reusable "as-is" for any other application that requires the same functionality.

In that sense, it is realistic to talk of the Digital Thread platform under construction as a large, heterogeneous, collaborative ecosystem where reusal is promoted well beyond the walled garden of a specific application domain. This is still a novelty to many, who are used to think in terms of their discipline and specialty, and see this possibility of direct reuse as an unexpected benefit, given that they are used to re-code rather than reuse.

In this respect, our value proposition sits clearly at the upper, application development layer, where we see the interoperability challenge truly reside.

We also see ourselves as systematic users of such pre-existing platforms, who are for us indeed welcome providers of Native DSLs. In this context, a number

of integrations in DIME relevant to the advanced manufacturing domain have already been addressed.

2.2 Low Code and DSLs

Low code development platforms enable their users to design and develop applications with minimal coding knowledge [27], with the support of drag-and-drop visual interfaces that operate on representations of code as encapsulated code wrappers. The main aim [24] of these platforms is to produce flexible, cost effective and rapid applications in a model driven way. Ideally, they are adaptive to enhancements and less complex is terms of maintenance. Model-driven development (MDD) is an approach to develop such systems using models and model refinement from the conceptual modelling phase to the automated model-to-code transformation of these models to executable code [19]. The main challenges with traditional software development approaches are the complexity in development at large scale, the maintenance over time, and the adaptation to dynamic requirements and upgrades [27]. Doing this on source code is costly, and it systematically excludes the application domain experts. who are the main knowledge and responsibility carriers. At the same time, the cost of quality documentation and training of new human resources for code-based development are other concerns in companies and organizations that depend on code.

Domain Specific Languages (DSLs) conveniently encapsulate most complexities of the underlying application domain. Encapsulation of code and abstraction to semantically faithful representations in models empowers domain experts to take advantage of these platforms. They can develop products in an efficient manner and also meet the growing demands of application development without having deep expertise in software development. Based on a study [2] from 451 researches, the maintenance effort with low code platforms proved to be 50–90% more efficient as compared to changes with classical coding languages.

3 Integration in DIME: A Quick Tutorial

The extension mechanisms of DIME concern two distinct levels: adding Services and adding Platforms.

- **Service integration** follows the *Native library* philosophy of DIME and can be carried out by following the steps described in Sect. 3.2 and Sect. 3.3. We illustrate this step by step on the basis of a REST integration, which is applicable also to the general microservice architectures.
- In **Platform integration** there is the additional challenge of the preparation of the platform in such a way that it addresses the needs of the underlying application. We will show this in the case of the R platform in Sect. 3.1.

In either case, the goal is to create adequate wrappers around the code one wishes to call or execute, an adequate call mechanism, and an appropriate signature representation of each of these SIBs within the DIME SIB Native library.

When the new SIB is utilised in an application, it is expected to communicate with the respective external service, technology, platform or infrastructure for the purposes of connection creation, system call, or movement of data. It is thus important that the runtime infrastructure is compatible, optimised and scalable in order to handle the required collection of data sets, instructions and generated results, e.g., strings, tables and graphs.

We describe now the three key steps of the DIME extension mechanism for the integration of external services or platforms, concerning the runtime infrastructure, the SIB declaration and the SIB implementation.

3.1 Runtime Infrastructure

Normally, the individual services are already optimised and deployed in some cloud service with a public or private access. The preparation of the platform is however still a challenge. Being DIME a web based application, the network latency can be reduced if the platform or technological infrastructure is deployed on the same network. The following steps are required to prepare a runtime docker container for the infrastructure.

1. Prepare and deploy a docker container for the desired technology or platform.
2. Get the network and connectivity details of the deployed container to be fed into DIME extended SIBs at runtime (or compile time, if the endpoint is constant).
3. Make sure to close all the connections after the successful communication with the client SIB in order to avoid dangling open connections.

Once this is done, we can define the new DSL in DIME via the SIB declaration mechanism. As an External native DSL is a collection of SIBs, we describe now how to add a SIB.

3.2 SIB Declaration

To develop a new (atomic) SIB in DIME, the SIB declaration is the first step: the declaration defines the SIB signature, with the data and control flow dependencies of the new SIB. The SIB declaration process is the same in both the Service and the Platform integration. To define a new atomic SIB,

1. Firstly, in the DIME-models section of the project explorer, we add a new empty file with extension ".sib"
2. This new file contains the signature of the new SIB. It starts with the keyword "sib", followed by the new SIB name, a colon and the path to the attached Java function. This is the function be invoked when the SIB will be used in the process modelling.
3. The subsequent lines contain the input parameters accompanied with the variable names and data types

```
sib sib_name : Java_file_path#function_name
    input_1: integer
    input_2: text
    -> control_branch_1
        output_1: text
    -> control_branch_2
```

Fig. 2. Sample SIB declaration signatures

4. Finally, considering the set of possible execution outcomes, a list of outgoing control branches is defined, one for each execution outcome. Each control branch starts with the symbol "—>", followed by the branch name and the output variable name and data type.

A sample SIB declaration signature is shown in Fig. 2.

When the DSL contains many SIBs, this declaration has to be done individually for each SIB.

3.3 SIB Implementation

Following the OTA philosophy, the two key considerations for SIB implementation are autonomy and modularity. The implementation of any SIB declared in Sect. 3.2 follows the following steps.

1. A Java file must be created (if not already available) under the "native-library" section of the "dependency" section in the project explorer, on the same path defined in the SIB declaration process.
2. In this file, write the implementation of all the Java functions against the respective signature declaration in the ".sib" file, i.e.,. matching the function name, return types and input data types.
3. Dependency management is an essential part of any project in order to interact correctly with any external service or platform. The services and platforms of interest normally expose some interface for interactions and system calls, frequently in form of APIs or drivers. The dependency management must be done both in the . POM file and in the header of the Java file of the project, to respectively download and import the drivers.
4. Finally, the Java implementation of the SIB could vary on the basis of the business logic of the application. This logic usually has three sections:
   - **Connection Establishment:** Normally, the first interaction with any external entity is the connection establishment. This happens by calling a constructor with the appropriate parameters, e.g., server name, IP, credentials.
   - **Function Call:** Once the connection is established, the subsequent logic may carry out with a sequence of system calls to send some parameters, data, code and instructions to the external entity, for it to act upon.

– **Result Parsing:** After the completion of the function/system calls, the result is returned in a raw format and must be parsed for the underlying platform to understand and use it appropriately according to control and data flow.

While this seems at first sight a lot, it is a simple and quite general mechanism. In the case of service integration, the runtime platform phase may not be needed, simplifying the task.

In the following section we exemplify the described procedure step by step for a platform and a service integration.

4 Case Studies: The R Platform and REST Services

We selected as examples the integration of the R platform and REST services, as they are good illustrations of the two main cases of native DSL integration.

4.1 R Integration as Platform Integration

R is a specialised language for statistical computing, optimised for data processing and analytics. We show now, how to integrate the R platform with DIME as a Native DSL and utilize its capabilities as drag-able SIBs.

Runtime Infrastructure: R is a different platform from Java, thus it requires a separate, independent execution infrastructure. First of all, a separate container must be deployed on docker to run a R infrastructure stack. Once the container is up and running, then we need to collect the connection details (container name, IP, and credentials) for the DIME app to be able to properly communicate. For this we use the docker command `build`, `run` and `network inspect`.

1. To build a new R infrastructure, open CLI, create a new directory and run the command:
   `docker build -t rserve Rserve/ --no-cache`
2. Once the image is built, deploy the image on the docker container using the command:
   `docker run --name EnvR -p 6311:6311 --rm rserve:latest`
3. Once the docker is deployed, then type the following command to get the IP and other network details of the image:
   `docker network inspect bridge .`

SIB Declaration: The SIB signatures for R (the function `plot_R_histogram` in this case) as shown in Fig. 3, consists of SIB keyword followed by its name and its I/O parameters. This signatures also contains the path of the Java function to be invoked whenever this SIB will be used in an application.

```
sib plot_R_histogram : info.App#plot_R_histogram
    file_name: text
    col_name: text
    title: text
    x_label: text
    y_label: text
    color: text
    -> success
        result: file
    -> noresult
    -> failure
```

Fig. 3. Sample SIB declaration signatures

1. Create a new DSL file under "dime-models" with the extension .sib, where multiple SIBs can be defined within the same domain.
2. Copy the same signatures from Fig. 3 to create your new SIB file.
3. Refresh your SIB explorer. The newly developed SIB must be visible in the SIB explorer as a draggable item.

SIB Implementation: For the SIB implementation we will do the following:

1. Create a Java file under the "native-library" section of "dependency" and write the Java function with the same name and parameters order as mentioned in the SIB signatures.
2. For the dependency management, copy the "REngine.Rserve" reference under the dependency tag in the POM file and import the Rserve libraries (RConnection, RFileInputStream, RFileOutputStream and RserveException) in your Java file.
3. To establish a connection with the R infrastructure, copy the R container IP and port number in the RConnection constructor. Once the connection is established, it will return a R_pointer, and at this point the R platform is ready to accept any command from this Java based DIME SIBs.
4. Move the dataset / CSV as BufferedInputStream from the DIME application to the R container using the established connection pointer, e.g., R_pointer.
5. Once the data is moved to the R server, the R commands can be sent from the DIME application to the R server as a suite of subsequent instructions to be executed on the R server on the already transported dataset. e.g. read data file, generate histogram with given parameters and name of output file handler. The R commands must be passed in double quotes as a parameter, inside the parseAndEval function referenced by the connection pointer. e.g. R_pointer.parseAndEval ("read.csv('a.csv')")
6. After the successful execution of subsequent R instructions, the generated result (in this case a histogram) is transported back as BufferedOutputStream, parsed into the DIME readable image and passed as a resulting data flow of the SIB using the getDomainFileController handler.

```
package app.demo
sib rest_read_str_list : file_path#Java_fn
    url : text
    input_var : text
    input : text
    output : text
    -> success
        output: [text]
    -> noresult
    -> failure
```

Fig. 4. SIBs explorer with the new Native SIBs

Following the same approach, the SIB declaration and SIB implementation process can be replicated to extend the R-DSL with any R capability.

4.2 RESTful Extension as Service Integration

RESTful services provide a great flexibility for communication with external systems through exposed APIs. The increasingly popular microservice architectures [23] are typically exposed as REST services. They play an important role at the enterprise level. The microservice paradigm helps design software services as suites of independently deployable components with the purpose of modularity, reusability and autonomy [23]. Different versions of these services may coexist in a system as a set of loosely coupled collaborative components and must be independently replaceable without impacting the operations of heterogeneous systems. We see now in detail how to integrate REST services along the template introduced.

Runtime Infrastructure: Services are normally provided by third parties and are already deployed on private or public servers. So they do not require any infrastructural preparation, unless we are developing and deploying our own REST services. We consider here a pre-built PHP based REST service that is already deployed on a public server and accessible via its URL.

SIB Declaration: The SIB declaration is shown in Fig. 4. We proceed as follows:

1. Create a new DSL file under "dime-models", with the extension .sib. Here, multiple SIBs can be defined within the same domain.
2. To create a new SIB, write the keyword `sib` followed by the SIB name, : (colon) and the path to the corresponding Java function.
   In subsequent lines, write the names of the input variables with their data types (in our case it is the URL of an external server), the input variable name and data type, and the output variable name where to retrieve the

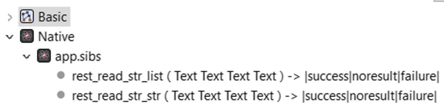

Fig. 5. SIBs explorer with the new Native SIBs

server response.
Finally, define the list of outgoing control branches based on the distinct outcomes, starting with symbol --->. In our case the three branches are "success", "noresult" and "failure".
3. Refresh your SIB explorer. The newly developed SIB must now be visible in the SIB explorer as a draggable item, as shown in Fig. 5.

SIB Implementation: For the SIB implementation we will do the following

1. Create a Java file under the "native-library" section of "dependency" and write a Java function with the same name and parameters order as mentioned in the SIB signatures.
2. For the dependency management, copy the "org.json" reference under the dependency tag in the POM file and import the HttpURLConnection and JSONObject libraries in your Java file.
3. To establish a connection with the REST service, copy the public URL and input parameters of the REST service and invoke the HttpURLConnection constructor followed by the getResponseCode() function.
4. On the successful response status, i.e. code 200, the service returns a JSON object that is further parsed into a Java readable string using the already imported JSONObject library functions, i.e. getString(JSON response).

Figure 6 shows the visual representation of the newly developed SIB as it appears when it is used in a process model. The required four inputs are being fed to this block using data flow (dotted) arrows. On success, the result will be conveyed as a string (or list of strings) to the successive SIB.

5 Conclusion and Discussion

In our Digital Thread efforts we are currently targeting primarily the application domain of advanced manufacturing including manufacturing analytics, and the data analytics field. In this context, data, processing and communications are expected to concern a large variety of devices, data sources, data storage technologies, communication protocols, analytics or AI technologies and tools, visualization tools, and more. This is where the ability to swiftly integrate external native DSLs plays a key role.

We presented therefore a generic extension mechanism for the integration of external services and platforms to DIME, an offline low-code IME, illustrating

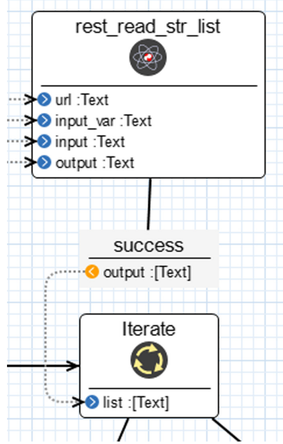

Fig. 6. The REST Read SIB in use: Visual representation in a model

in detail the technique on the basis of the R platform for data analytics and a REST service.

We used DIME's native library mechanism, with signature declaration, linked Java backend code, and where the code is merged with the logic layer at compile time. In previous work [4] we already showed how to address also the integration in Pyrus, an online no-code graphical data analytics tool based on Pyro [28] and linked with Jupyter Hub for functions discovery and code execution also discussed in detail in [29].

The Pyrus integration is simpler as it is tightly connected with Jupyter Hub. For example, to display new python functions as components in Pyrus, custom signatures are added to the python files defined in Jupyter Hub, and the data flow pipeline of the service is modelled in the Pyrus frontend.

In comparison with prior integration techniques, e.g. in jABC [11,15], ETI [13], jETI [12] and jABC/DyWA [22], this is a much simpler mechanism, and it follows much more tightly the One Thing Approach philosophy of integration [17].

As shown in Fig. 1, the span of External native DSLs we are currently building is quite impressive. It will cover proactive maintenance and building automation applications in the collaborations in Confirm, devoted to Smart Manufacturing, an open source biomechanical prosthetic socket design and optimization platform based on Gibbon [20] in Lero, analytics for the Digital Humanities [10], and various healthcare applications in the context of the Limerick Cancer Research Centre. In this sense we are hopeful to indeed bootstrap the effect of reuse of many DLSs and also processes across several application domains. This way, we also wish to showcase the power of model driven low code application development on real life examples from research and industrial applications.

Acknowledgment. This work was supported by the Science Foundation Ireland grants 16/RC/3918 (Confirm, the Smart Manufacturing Research Centre) and 13/RC/2094_2 (Lero, the Science Foundation Ireland Research Centre for Software).

References

1. Edgex foundry: The edgex foundry platform. https://www.edgexfoundry.org/. Accessed July 2021
2. Research: Intelligent process automation and the emergence of digital automation platforms. https://www.redhat.com/cms/managed-files/mi-451-research-intelligent-process-automation-analyst-paper-f11434-201802.pdf. Accessed February 2021
3. Boßelmann, S., et al.: DIME: a programming-less modeling environment for web applications. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 809–832. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_60
4. Chaudhary, H.A.A., Margaria, T.: Integration of micro-services as components in modeling environments for low code development. Proc. ISP RAS **33**(4) (2021)
5. Gartner: Gartner forecasts worldwide low-code development technologies market to grow 23% in 2021. https://www.gartner.com/en/newsroom/press-releases/2021-02-15-gartner-forecasts-worldwide-low-code-development-technologies-market-to-grow-23-percent-in-2021. Accessed February 2021
6. iBASEt: The digital thread explained. https://www.ibaset.com/the-digital-thread-explained/. Accessed July 2021
7. John, J., Ghosal, A., Margaria, T., Pesch, D.: DSLS and middleware platforms in a model driven development approach for secure predictive maintenance systems in smart factories. In: Margaria, T., Steffen, B. (eds.) ISoLA 2021, LNCS, vol. 13036, pp. 146–161, Springer, Heidelberg (2021)
8. John, J., Ghosal, A., Margaria, T., Pesch, D.: Dsls for model driven development of secure interoperable automation systems. In: 2021 Forum for Specification and Design Languages (FDL). IEEE (2021, September (in print))
9. Jorges, S., Kubczak, C., Pageau, F., Margaria, T.: Model driven design of reliable robot control programs using the jabc. In: Proceedings EASe'07, vol. 07, pp. 137–148 (2007). https://doi.org/10.1109/EASE.2007.17
10. Khan, R., Schieweck, A., Breathnach, C., Margaria, T.: Historical civil registration record transcription using an extreme model driven approach. Proc. ISP RAS **33**(3) (2021)
11. Kubczak, C., Margaria, T., Steffen, B., Nagel, R.: Service-oriented Mediation with jABC/jETI (2008)
12. Lamprecht, A.L., Margaria, T., Steffen, B.: Bio-jETI: a framework for semantics-based service composition. BMC Bioinform. **10**(Suppl 10), S8 (2009). https://doi.org/10.1186/1471-2105-10-S10-S8
13. Margaria, T.: Web services-based tool-integration in the ETI platform. Softw. Syst. Model. **4**(2), 141–156 (2005). https://doi.org/10.1007/s10270-004-0072-z
14. Margaria, T., Chaudhary, H.A.A., Guevara, I., Ryan, S., Schieweck, A.: The interoperability challenge: building a model-driven digital thread platform for CPS. In: Margaria, T., Steffen, B. (eds.) ISoLA 2021, LNCS, vol. 13036, pp. 393–413. Springer, Heidelberg (2021)

15. Margaria, T., Nagel, R., Steffen, B.: Remote integration and coordination of verification tools in JETI. In: Proceedings of the 12th IEEE International Conference on the Engineering of Computer-Based Systems, pp. 431–436. IEEE Computer Society, Los Alamitos, CA, USA (2005). https://doi.org/10.1109/ECBS.2005.59
16. Margaria, T., Schieweck, A.: The digital thread in Industry 4.0. In: Ahrendt, W., Tapia Tarifa, S.L. (eds.) IFM 2019. LNCS, vol. 11918, pp. 3–24. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34968-4_1
17. Margaria, T., Steffen, B.: Business process modeling in the jabc: the one-thing approach. In: Handbook of Research on Business Process Modeling, pp. 1–26. IGI Global (2009)
18. Margaria, T., Steffen, B.: Extreme model-driven development (xmdd) technologies as a hands-on approach to software development without coding. In: Encyclopedia of Education and Information Technologies, pp. 732–750 (2020)
19. Mellor, S.J., Clark, T., Futagami, T.: Model-driven development: guest editors' introduction. IEEE Softw. **20**(5), 14–18 (2003). issn 0740–7459
20. Moerman, K.M.: Gibbon: the geometry and image-based bioengineering add-on. J. Open Source Softw. **3**(22), 506 (2018)
21. Naujokat, S., Lybecait, M., Kopetzki, D., Steffen, B.: CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. Int. J. Softw. Tools Technol. Transfer **20**(3), 327–354 (2017). https://doi.org/10.1007/s10009-017-0453-6
22. Neubauer, J., Frohme, M., Steffen, B., Margaria, T.: Prototype-driven development of web applications with DyWA. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8802, pp. 56–72. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45234-9_5
23. Newman, S.: Building microservices: designing fine-grained systems. O'Reilly Media, Inc. (2015)
24. Sanchis, R., García-Perales, Ó., Fraile, F., Poler, R.: Low-code as enabler of digital transformation in manufacturing industry. Appl. Sci. **10**(1), 12 (2020)
25. Steffen, B., Margaria, T., Claßen, A., Braun, V.: The METAFrame'95 environment. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 450–453. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61474-5_100
26. Steffen, B., Margaria, T., Claßen, A., et al.: Heterogeneous analysis and verification for distributed systems. In: Software-Concepts and Tools, pp. 13–25 (1996)
27. Waszkowski, R.: Low-code platform for automating business processes in manufacturing. IFAC-PapersOnLine **52**(10), 376–381 (2019)
28. Zweihoff, P., Naujokat, S., Steffen, B.: Pyro: generating domain-specific collaborative online modeling environments. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 101–115. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_6
29. Zweihoff, P., Steffen, B.: Pyrus: an online modeling environment for no-code data-analytics service composition. In: Margaria, T., Steffen, B. (eds.) ISoLA 2021, LNCS, vol. 13036, pp. 18–40. Springer, Heidelberg (2021)