Ref. Ares(2016)528784 - 01/02/2016

Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

DICE simulation tools - Initial version

Deliverable 3.2

Deliverable:	D3.2
Title:	DICE simulation tools - Initial version
Editor(s):	Abel Gómez (ZAR)
Contributor(s):	Simona Bernardi (ZAR), Giuliano Casale (IMP), Abel Gómez (ZAR), Shuai Jiang (IMP) and José Merseguer (ZAR)
Reviewers:	Darren Whighman (FLEX), Marc Gil (PRO)
Type (R/P/DEC):	Report
Version:	1.1
Date:	1-February-2016
Status:	Final version
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2015, DICE consortium – All rights reserved

The DICE project (February 2015-January 2018) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644869

Executive summary

This document presents the initial results of the development of the *Simulation Tools* and describes the relationships with previous deliverables – mainly D1.2, which presents the requirements and use case scenarios –. This deliverable also serves as a baseline for upcoming deliverables – mainly D1.3 (*Architecture definition and integration plan, initial version*, to be released in M12 too) and D1.4 (*Architecture definition and integration plan, final version*). Additionally, this document provides a comprehensive description of the *Simulation Tool*, currently as a prototype, its architecture and the interactions among its internal components.

All the artifacts presented in this document are publicly available in the so-called *DICE-Simulation* Repository [**dice:simulation:repo**], whose structure and components are described in the Appendix A of this document.

Glossary

DIA	Data-Intensive Application
DICE	Data-Intensive Cloud Applications with iterative quality enhancements
IDE	Integrated Development Environment
M2M	Model-to-model Transformation
M2T	Model-to-text Transformation
MARTE	Modeling and Analysis of Real-time Embedded Systems
MDE	Model-Driven Engineering
OSGi	Open Services Gateway initiative
PNML	Petri Net Markup Language
QVT	Meta Object Facility (MOF) 2.0 Query/View/Transformation Standard
QVTc	QVT Core language
QVTo	QVT Operational Mappings language
QVTr	QVT Relations language
UML	Unified Modelling Language

Contents

List of Figures

List of Tables

1 Introduction and Context

The focus of the DICE project is to define a quality-driven framework for developing data-intensive applications that leverage Big Data technologies hosted in private or public clouds. DICE offers a novel profile and tools for data-aware quality-driven development. DICE-profiled models are fed into a set of simulation, analysis and optimization tools to obtain high-quality applications. One of these tools within the DICE framework is the so-called *Simulation Tool*, which allows evaluating quality properties of data-intensive applications, in particular efficiency and reliability metrics. This document describes the initial version of the *Simulation Tool* prototype, developed in the scope of WP3 as Task 3.2, and published as an open source tool in the *DICE-Simulation* repository [**dice:simulation:repo**].

1.1 Objectives of WP3

The goal of WP3 is to develop a quality analysis tool-chain that will be used to guide the early design stages of the data intensive application and guide quality evolution once operational data becomes available. The main outputs of these tasks are tools for simulation-based reliability and efficiency assessment, for formal verification of safety properties related to the sequence of events and states that the application undergoes, and numerical optimization techniques for search of optimal architecture designs.

In WP3 are also defined Model-to-model (M2M) transformations that accept as input design models defined in T2.1 and T2.2, and produce as outputs the analysis models used by the quality tools.

1.2 Objectives of Task 3.2

Task 3.2 focus on the development of a hybrid simulation framework that combines black-box and white-box models, to evaluate quality properties of data-intensive applications, in particular efficiency and reliability metrics. White-box models based on colored Stochastic Petri nets are used to describe the abstract properties of application models developed according to the DICE profile, which is proposed in WP2.

Colored Stochastic Petri nets are good abstractions for data-intensive applications, since a token circulating in the model represents a request being processed and atomic fork/join operations and colors can be easily used to express at the same time the memory, disk read/write operations, network/stream traffic and other concurrent operations that a single request implies on the resources.

A second major challenge addressed by this task is the inclusion in simulation of black-box models to describe the execution characteristics of hosted Big Data services, such as Amazon Elastic MapReduce. Where possible, techniques to accelerate the evaluation of rare events (e.g., failures) will be integrated in the simulation framework, in order to reduce the time needed to accurately assess reliability metrics.

1.3 Objectives of this Document

This document serves as an initial demonstration of the tools to be developed within Task 3.2. Specifically, this demonstrator provides a fully-working prototype of the *DICE Simulation Tool*. This prototype is able to cover all the steps of the simulation workflow (i.e., model, transform, simulate, retrieve results).

This document also provides an architectural and behavioral description of the tool, serving as a baseline for D1.3 and D1.4 (i.e., *Architecture definition and integration plan*, initial version and final version, respectively).

1.4 Structure of the Document

The structure of this deliverable is as follows:

- Section 2 summarizes the involved actors, use cases and requirements that Task 3.2 aims to cover.
- Section 3 presents the proposed tool architecture and the interactions among its internal components.
- Section 4 shows the current prototype, its interface from the users' point of view and its usage.

- Section 5 summarizes the goals achieved, and outlines the future work.
- Appendix A provides details on the *Simulation Tool repository*.

2 Requirements and Usage Scenarios

Deliverable D1.2 [**dice:d1.2**, **dice:d1.2:companion**], released on month 6, presented the requirements analysis for the DICE project. The outcome of the analysis was a consolidated list of requirements and the list of use cases that define the project's goals that guide the DICE technical activities.

This section recapitulates, for Task T3.2, these requirements and use case scenarios and explains how they have been fulfilled in the current *Simulation Tool* prototype.

2.1 Tools and Actors

As specified in D1.2, the data-aware quality analysis aims at assessing quality requirements for DIAs and at offering an optimized deployment configuration for the application. The assessment starts from the DIA UML design, which includes not only the functionality of the system but also the quality requirements and corresponding parameters. The assessment is accomplished making use of the following tools:

Transformation Tools — These tools take as input a UML DICE-profiled design representing a DIA and produce suitable formal models.

Simulation Tools — The simulation tools take as input the models produced by the *Transformation Tools* and validate the performance and reliability requirements of the DIA.

Verification Tools — The verification tools aim at checking the so-called safety properties for the DIA.

Optimization Tools — The optimization tools apply at DDSM level and evaluate the corresponding Petri net models for deciding which deployment is the optimal one regarding to a predefined criteria.

In the remaining of this document, we will focus on tools related to Tasks T3.1 and T3.2, i.e., the *Transformation Tools* and the *Simulation Tools*. Regarding these tools, D1.2 specifies that the following stakeholders use them directly:

QA Engineer — The application quality engineer uses the *Simulation Tools* through the DICE IDE.

Simulation Tools — The *Transformation Tools* are not used by human actors directly but internally for the rest of the WP3 tools.

2.2 Use Cases

The requirements elicitation of D1.2 considers a single use case[1] that concerns the *Simulation Tools* component, the UC3.1. This use case can be summarized as[2]:

ID	UC3.1
Title	Verification of reliability or performance properties from a DPIM/DTSM DICE annotated UML model
Priority	Required
Actors	*QA Engineer*, *IDE*, *Transformation Tools*, *Simulation Tools*
Pre-conditions	There exists a DPIM/DTSM level UML annotated model.
Post-conditions	The *QA Engineer* gets information about the predicted metric value in the technological environment being studied

[1] UC3.1.1 (*Verification of throughput from a DPIM DICE annotated UML model*) is a specialization of UC3.1, and as such will not be considered in the present document

[2] For detailed information, refer to the *Requirement Specification* document [**dice:d1.2**]

2.3 Requirements

To support the previous use case scenario of the *Simulation Tools* component, the following (summarized) requirements were defined:

ID	R3.1
Title	M2M Transformation
Priority	Must have
Description	The *Transformation Tools* MUST perform a model-to-model transformation taking the input from a DPIM or DTSM DICE annotated UML model and returning a formal model [. . .].

ID	R3.2
Title	Taking into account relevant annotations
Priority	Must have
Description	The *Transformation Tools* MUST take into account the relevant annotations in the DICE profile [. . .] and transform them into the corresponding artifact [. . .]

ID	R3.3
Title	Transformation rules
Priority	Could have
Description	The *Transformation Tools* MAY be able to extract, interpret and apply the transformation rules from an external source.

ID	R3.4
Title	Simulation solvers
Priority	Must have
Description	The *Simulation Tools* will select automatically [. . .] the right solver [. . .]

ID	R3.5
Title	Simulation of hosted big data services
Priority	Must have
Description	The *Simulation Tools* MUST be able to describe the execution characteristics of hosted big data services.

ID	R3.6
Title	Transparency of underlying tools
Priority	Must have
Description	The *Transformation Tools* and *Simulation Tools* MUST be transparent to users. [. . .]

ID	R3.10
Title	SLA specification and compliance
Priority	Must have
Description	[. . .] *Simulation Tools* [. . .] MUST permit users to check their outputs against SLA's included in UML model annotations. [. . .]

ID	R3.13
Title	White/black box transparency
Priority	Must have
Description	For the *Transformation Tools* and the *Simulation Tools* there will be no difference between white box and black box model elements.

ID	R3.14
Title	Ranged or extended what if analysis
Priority	Could have
Description	The *Simulation Tools* will be able to cover a range of possible values for a parameter and run a simulation for every different scenario [. . .]

ID	R3IDE.1
Title	Metric selection
Priority	Must have
Description	The DICE IDE MUST allow to select the metric to compute from those defined in the DPIM/DTSM DICE annotated UML model. [. . .]

ID	R3IDE.2
Title	Timeout specification
Priority	Should have
Description	The IDE SHOULD allow the user to set a timeout and a maximum amount of memory to be used when running the *Simulation Tools* and the *Verification Tools*. [. . .]

ID	R3IDE.3
Title	Usability
Priority	Could have
Description	The *Transformation Tools* and *Simulation Tools* MAY follow some usability, ergonomics or accesibility standard [. . .]

ID	R3IDE.4
Title	Loading the annotated UML model
Priority	Must have
Description	The DICE IDE MUST include a command to launch the *Simulation Tools* [. . .] for a DICE UML model that is loaded in the IDE

3 The Simulation Tools

Use case UC3.1 specifies that, from *an existing DPIM/DTSM level UML annotated model* (pre-condition), *the* QA Engineer *gets information about the predicted metric value in the technological environment being studied* (post-condition).

To obtain such information, the following steps need to be performed:

1. The *QA Engineer* models a DPIM/DTSM model applying the DICE profile to a UML model using the DICE-IDE.
2. The *QA Engineer* starts a new simulation using the DICE-profiled UML models as input.
3. The DICE-profiled UML models are translated within the simulation process to formal models, which can be automatically analysed, using M2M and M2T transformations.
4. The simulation process is configured, specifying the kind of analysis to perform and the additional input data required to run the analysis.
5. The simulation process is executed, i.e., the formal models are analysed using open-source evaluation tools (in particular *GreatSPN* [**greatspn**] and *Java Modelling Tools* [**JMT**]).
6. The result produced by the evaluation tool is processed to generate a tool-independent report, conformant to a report model, with the assessment of performance and reliability metrics.
7. The tool-independent report is fed into the DICE-IDE and is shown to the user in the GUI.

From this description, we can identifiy the following core components:

The DICE-IDE is an integrated development environment used by the *QA Engineer* to develop DPIM/DTSM models. It provides to the end-users all the functionality provided by the DICE framework.

The Simulator is a DICE component in charge of executing the simulation.

The M2M Transformation Engine executes model-to-model transformations. It handles transformations between technical spaces[3].

The M2T Transformation Engine executes model-to-text transformations. It handles tool-specific transformations within the same technical space.

The Evaluation Tool (e.g. *GreatSPN*) performs an evaluation of a specific formal model (e.g., a Petri net).

3.1 Components Interaction

Based on the previous core components, we have modeled their interactions as depicted in Fig. 1. For the sake of maintainability, the *Simulator* component has been split up in UI and non-UI components (i.e., *Simulator-GUI* and *Simulator* respectively).

Specifically, the sequence diagram depicted in the Fig. 1 describes the specific steps to simulate a DICE-profiled UML diagram using for example the *GreatSPN* tool as the underlying evaluation tool.

As it can be seen in the figure, the modeling step is outside the scope of the *Simulation* phase, and the model to be analysed is supposed to pre-exist and is managed by the *DICE-IDE*. When the user wants to simulate a model, he/she invokes the *Simulator-GUI*, which parses the model and asks the user any additional required information. When this information is obtained, the *Simulator-GUI* calls the *Simulator* that will handle the simulation in background.

[3] A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities [**kurtev:tech˙spaces**].

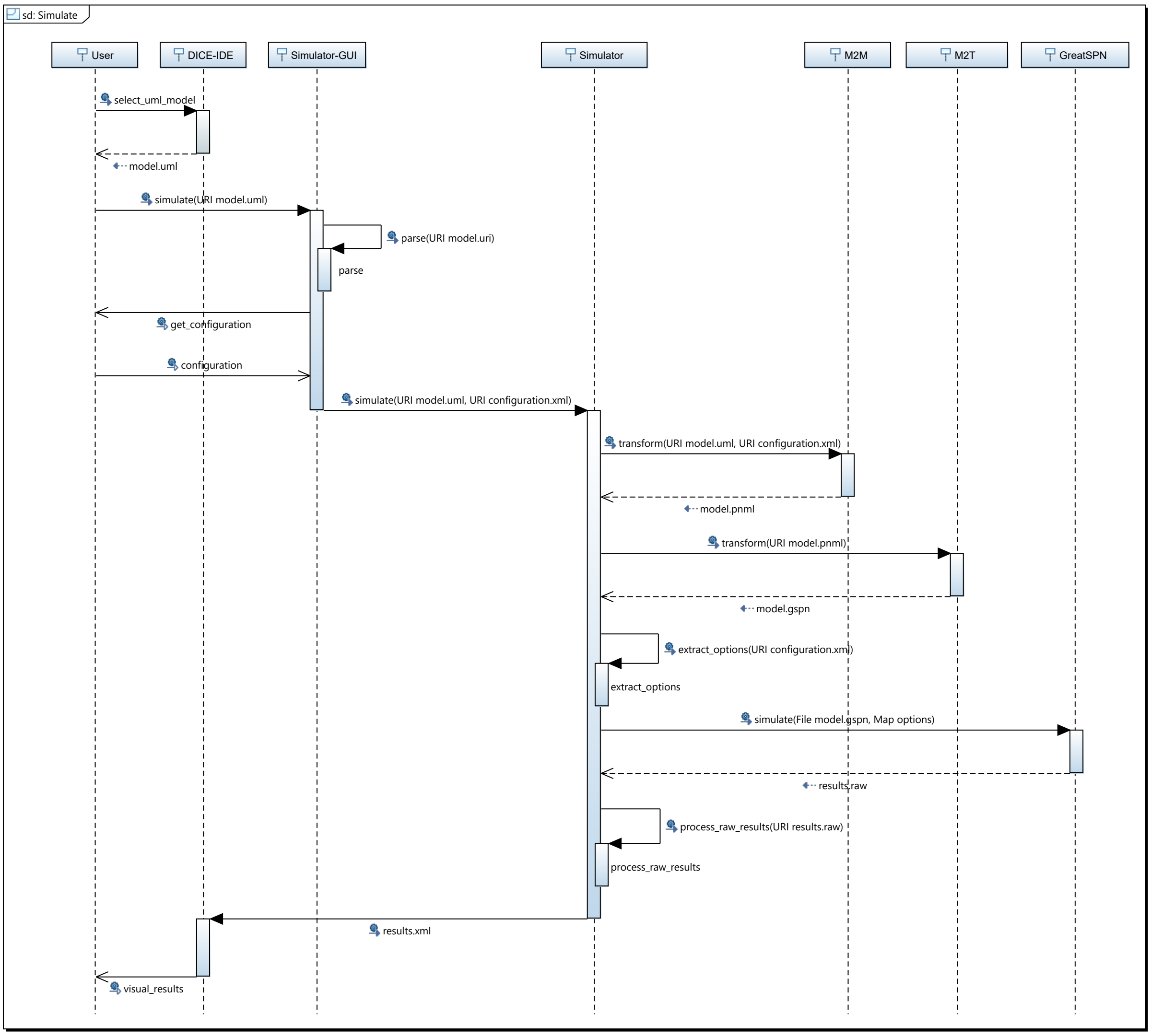

Figure 1: Sequence diagram depicting the interactions between the components of the *Simulation Tools*

The *Simulator* will then orchestrate the interaction among all the different modules: first, the M2M transformation module will create an intermediate representation, which for GreatSPN is the PNML [**pnml:primer**] representation of the DICE-profiled model; second, the PNML file will be transformed to a GreatSPN-specific Petri net description file; third, the *Simulator* will start the analysis of the Petri net using *GreatSPN*; and finally, when the analysis ends, the raw results produced by *GreatSPN* will be converted into a formatted results file. These formatted results will be processed by the *DICE-IDE* that will show them to the user in a visual form.

3.2 Tool Architecture

Figure 2 shows the simplified architecture of the *Simulation Tools* and the internal data flows. This figure depicts the actual modules that implement the sequence diagram shown in Fig. 1 and realize the use case UC3.1 described in Section 2.

Next, we provide a description of the different modules, the data they share, and their nature:

1. The *DICE-IDE* is an Eclipse-based environment in which the different components are integrated.

2. A *simulation process* starts by defining a set of DICE-Profiled UML models. For this stage, a pre-existing modeling tool is used.

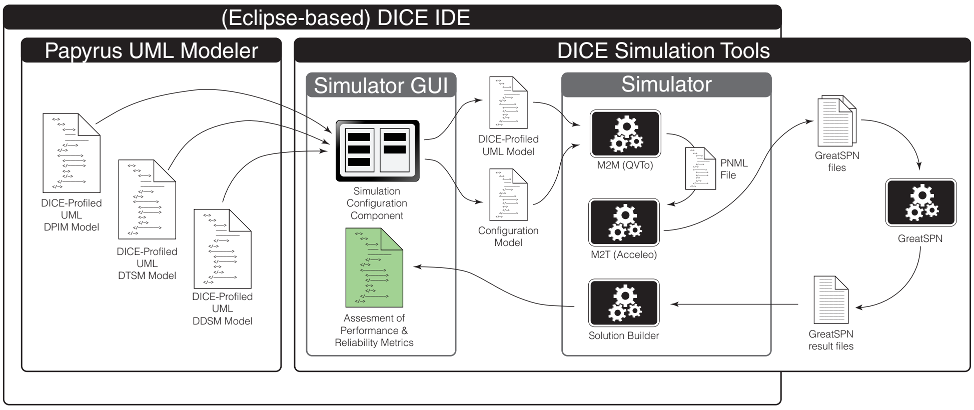

Figure 2: High-level architecture of the *Simulation Tools*

Papyrus UML [**papyrus:starters**] is one of the open source UML modeling tools that support the MARTE (Modeling and Analysis of Real-time Embedded Systems) profile [**omg:marte**], in which the DICE profile is based on. As proposed in the Deliverable D1.1 (*State of the art analysis*) [**dice:d1.1**], this component/tool is used to perform the initial modeling stage.

3. When the user (the *QA Engineer*) wants to simulate a model, he/she uses the *Simulator GUI* to start a simulation.

   The **Simulator GUI** is an ad hoc Eclipse component that contributes a set of graphical interfaces to the DICE-IDE. These interfaces are tighly integrated within the DICE-IDE providing a transparent way for interacting with the underlying analysis tools.

   The **Simulation Configuration Component** is a sub-component of the *Simulator GUI*. It is in charge of: (i) asking for the model to be simulated (using the DICE-IDE infrastructure, dialogs, etc.); and (ii) asking for any additional data required by the *Simulator*.

4. When the user has finished the configuration of a simulation, the *Configuration Tool* passes two different files to the *Simulator*: the *DICE-profiled UML model* (i.e., the model to be analysed) and the *Configuration model*.

   The **Simulator** is an ad hoc OSGi component that runs in background. It has been specifically designed to orchestrate the interaction among the different tools that perform the actual analysis.

5. The *Simulator* executes the following steps: (i) transforms the UML model into a PNML file using a M2M transformation tool; (ii) converts the previous PNML file to a GreatSPN-readable file using a M2T transformation tool; (iii) evaluates the GreatSPN-readable file using the GreatSPN tool; and (iv) builds a tool-independent solution from the tool-specific file produced by *GreatSPN*.

   To execute the **M2M** transformations we have selected the eclipse **QVTo** transformations engine. QVT [**omg:qvt**] is the standard language proposed by the OMG (the same organism behind the UML and MARTE standards) to define M2M transformations. QVT proposes three possible languages to define model transformations: *operational mappings* (QVTo, imperative, low-level), *core* (QVTc, declarative, low-level) and *relations* (QVTr, declarative, high-level). However, although there are important efforts to provide implementations for all of them, only the one for QVTo is production-ready, and as such is the chosen one.

   To execute the **M2T** transformations we have selected **Acceleo** [**acceleo**]. Starting from Acceleo 3, the language used to defined an Acceleo transformation is an implementation of the MOFM2T standard [**omg:mtl**], proposed by the OMG too. In this sense, we have selected Acceleo to make all our toolchain compliant to the OMG standards, from the definition of the initial (profiled) UML models to the 3rd party analysis tools (which use a proprietary format).

The analysis is performed using the **GreatSPN** tool. GreatSPN is a complete framework for the modeling, analysis and simulation of Petri nets. This tool can leverage those classes of Petri nets needed by our simulation framework, i.e., Generalized Stochastic Petri Nets (GSPN) and their colored version, namely Stochastic Well-formed Nets (SWN). GreatSPN includes a wide range of GSPN/SWN solvers for the computation of performance and reliability metrics (the reader can refer to the ”State of the art analysis” deliverable D1.1 for details about the GreatSPN functionalities).

6. Finally, the *tool-independent report* produced by the *Simulator* is presented in the *DICE-IDE* using a graphical component of the *Simulator GUI*. This component provides a comprehensive *Assesment of Performance and Reliability Metrics* report in terms of the concepts defined in the initial UML model.

4 Tool Overview and Usage

This section shows what the *Simulation Tool* looks like from the users' point of view, and provides a quick description on how to use it in combination with GreatSPN.

First, it is worth to recall that the modeling phase is done using Papyrus. Since there exists extensive documentation on how to use this tool to create profiled UML models [**papyrus:starters**, **papyrus:profiles**, **papyrus:activity**, **papyrus:collab**, **papyrus:sequence**, **papyrus:stylesheets**], we will not provide details on the usage of this specific tool.

Figure 3 shows a general view of the *Papyrus* modeling perspective. On the left of the figure, the different explorers (*Project Explorer*, *Model Explorer* and *Outline*) are shown. The rest of the figure shows the *Model Editor* and the *Properties* view.

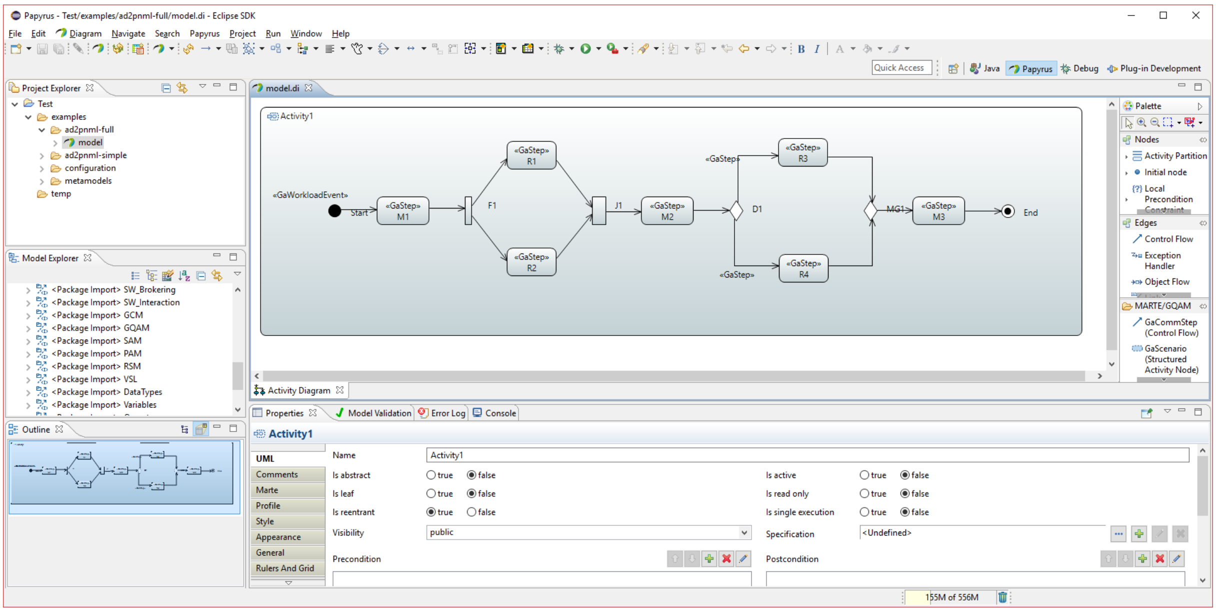

Figure 3: General view of the *Papyrus* modeling perspective in Eclipse/DICE-IDE

The model itself is depicted in the canvas of the *Model Editor*. Profiles, stereotypes and tagged values are defined using the *Properties* view.

Figs. 4 and 5 show in the *Properties* view some tagged values (of the *GaStep* MARTE stereotype) that are applied to some model elements. Specifically, Fig. 4 shows the *host demand* tagged value of the *M1* element (defined as `(value=0.5,unit=s,statQ=mean,source=assm)`[4]), while Fig. 5 shows the *prob* tagged value of the control flow between *M2* and *R3* (`(value=$p1,source=assm)`). As it can be seen, a variable (`$p1`) has been used for the latter (we will explain more on variables later on this section).

[4] *Tagged values* are specified in Papyrus-MARTE using the so-called *Value Specification Modeling* language. Details on this language can be found on the MARTE Standard [**omg:marte**].

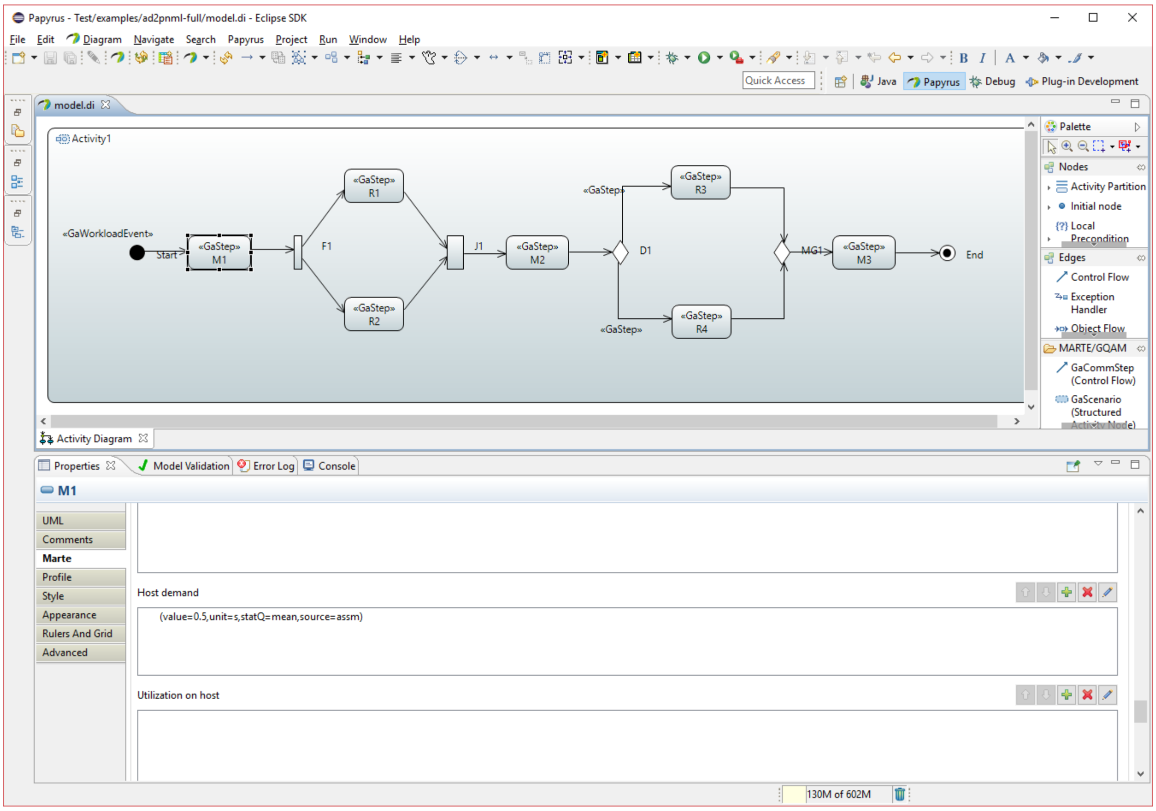

Figure 4: *Host demand* tagged value of the *M1* element, prototyped as *GaStep* from *MARTE*

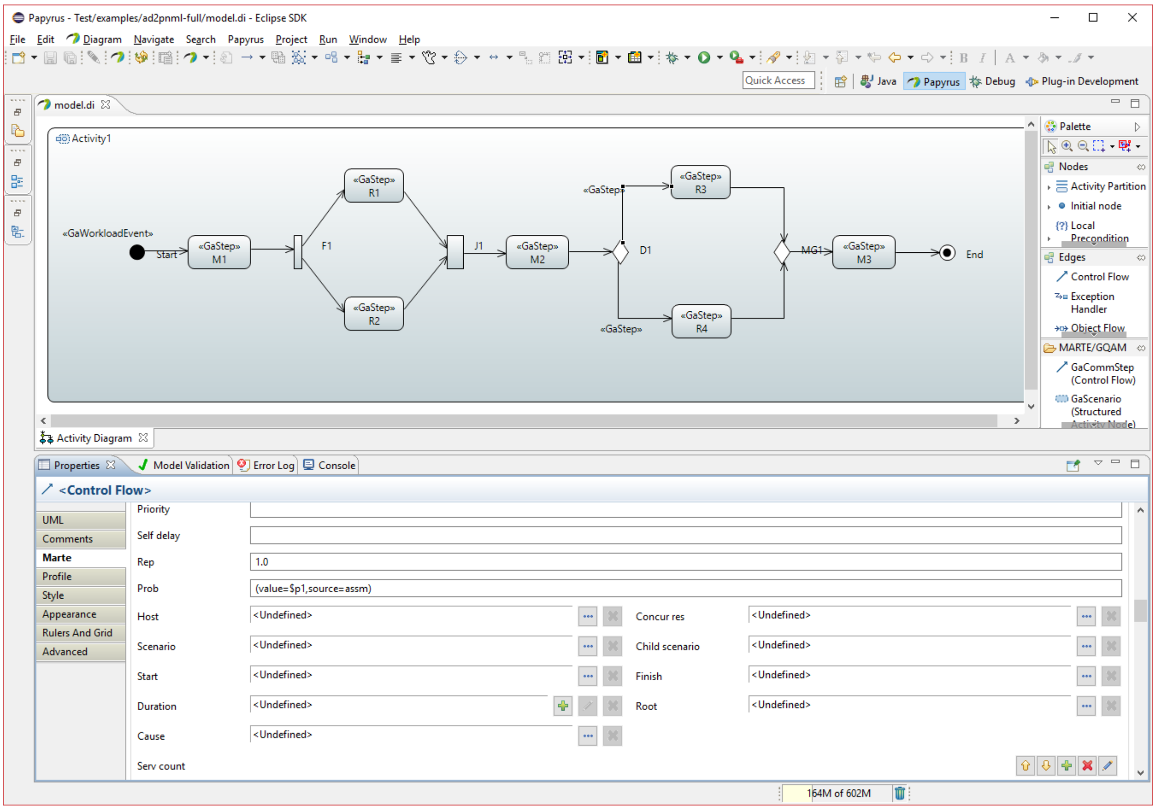

Figure 5: *Prob* tagged value of the control flow between *M2* and *R3*, prototyped as *GaStep* from *MARTE*

Figure 6 shows the drop-down button that is used to open the *Launch Configurations...* window, in which the *Simulator GUI* has been integrated.

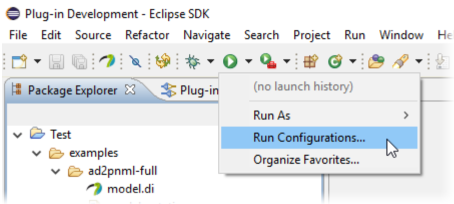

Figure 6: Open the *Run Configurations...* window

Figure 7 shows how a new *launch configuration* – which controls a *Simulation process* – is created from scratch, while Fig. 8 shows what the *Simulator GUI (Configurator module)* looks like.

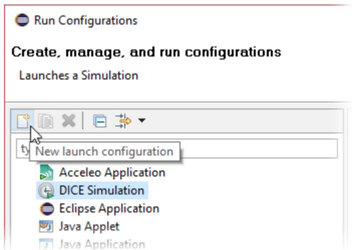

Figure 7: Create a new *Simulation launch configuration*

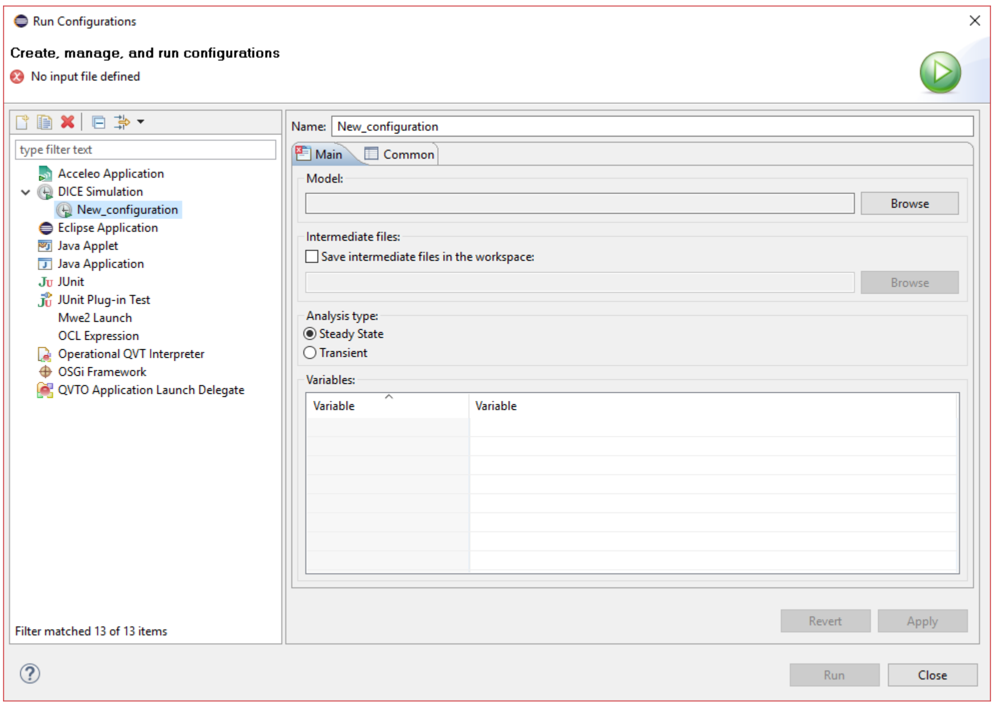

Figure 8: The *Run Configurations* window showing the *Simulator GUI (Configurator module)*

It is possible to directly create a new *launch configuration* from an existing model using the contextual menu shown in Fig. 9.

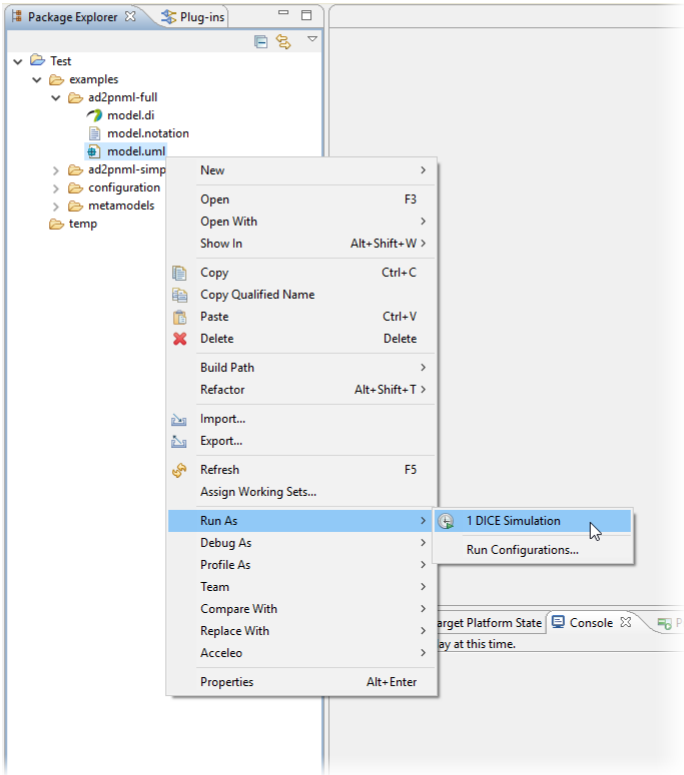

Figure 9: Create a new *Simulation launch configuration* from a workspace model

The pre-configured launch configuration that is created is shown in Fig. 10.

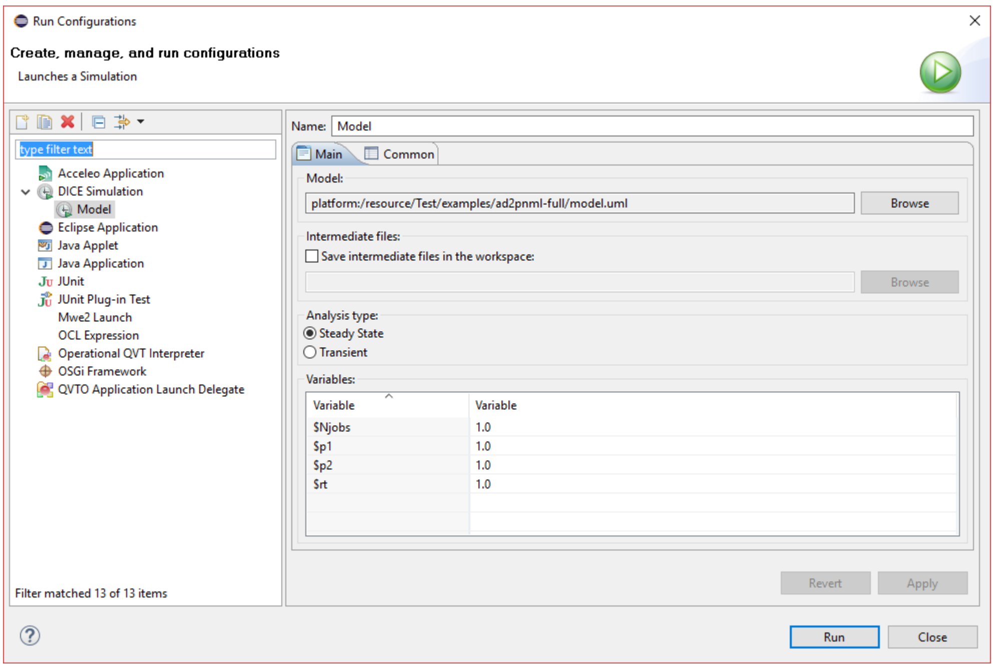

Figure 10: A newly created *Simulation launch configuration* with the intial values

As it can be seen, the *launch configuration* is initialized with the input model. This model is analysed searching for variables that need to be initialized. The table shown at the bottom of the figure is then used to customize the variables' values.

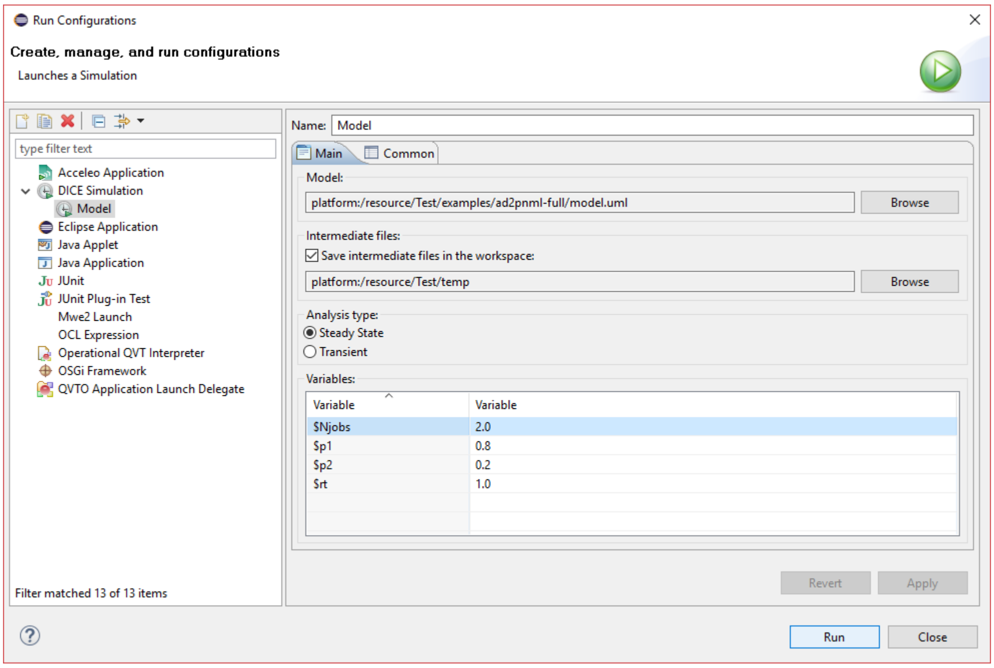

Figure 11: A *Simulation launch configuration* ready to be executed

Figure 11 shows a *launch configuration* ready to be executed. In this case, all the intermediate files will be saved in the workspace. This is, however, only an option since all the intermediate transformation steps are executed in a transparent way. Once the user clicks on the *Run* button, the simulation starts.

The simulation can be tracked and controlled using the *Debug* perspective as shown in the Fig. 12. In the figure, two key *views* can be identified: the *Debug* view and the *Console* view. The former shows information about the *Simulation process* (identifier, state, exit value, etc.); while the latter shows the messages that the simulation process dumps into the standard out and the standard error streams. In the case of *GreatSPN*, these messages allow to monitor the accuracy achieved by the running process and the number of simulation steps that have been already performed.

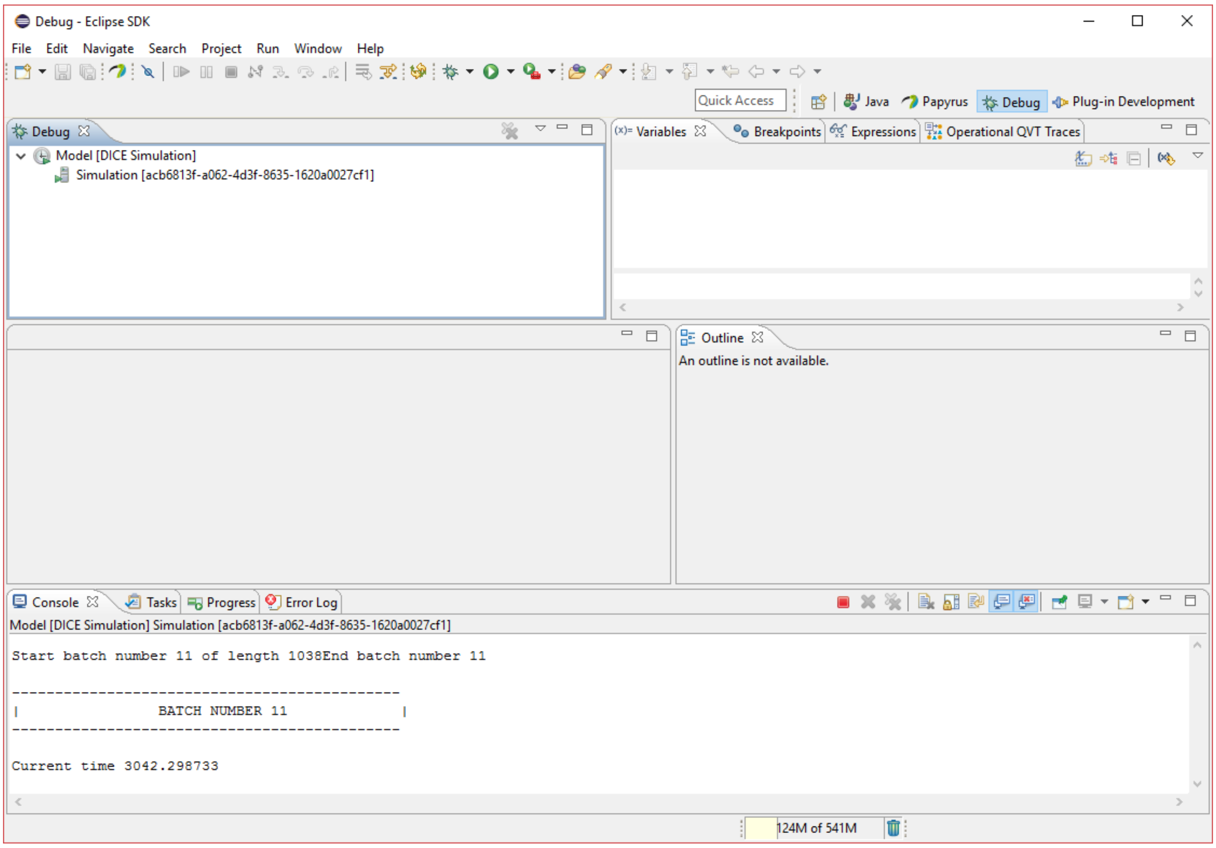

Figure 12: Running a simulation in the *Debug* perspective

As Fig. 13 shows, it is also possible to stop the simulation process at any moment by using the *Stop* button of the GUI.

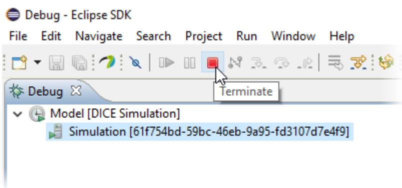

Figure 13: Detail of the stop button, which force terminates a simulation

When the simulation finishes, the user can still access the simulation console and the simulation process information (until he/she cleans the *Debug* view using the  button). As Fig. 14 shows, the simulation process has finished correctly (exit value is 0).

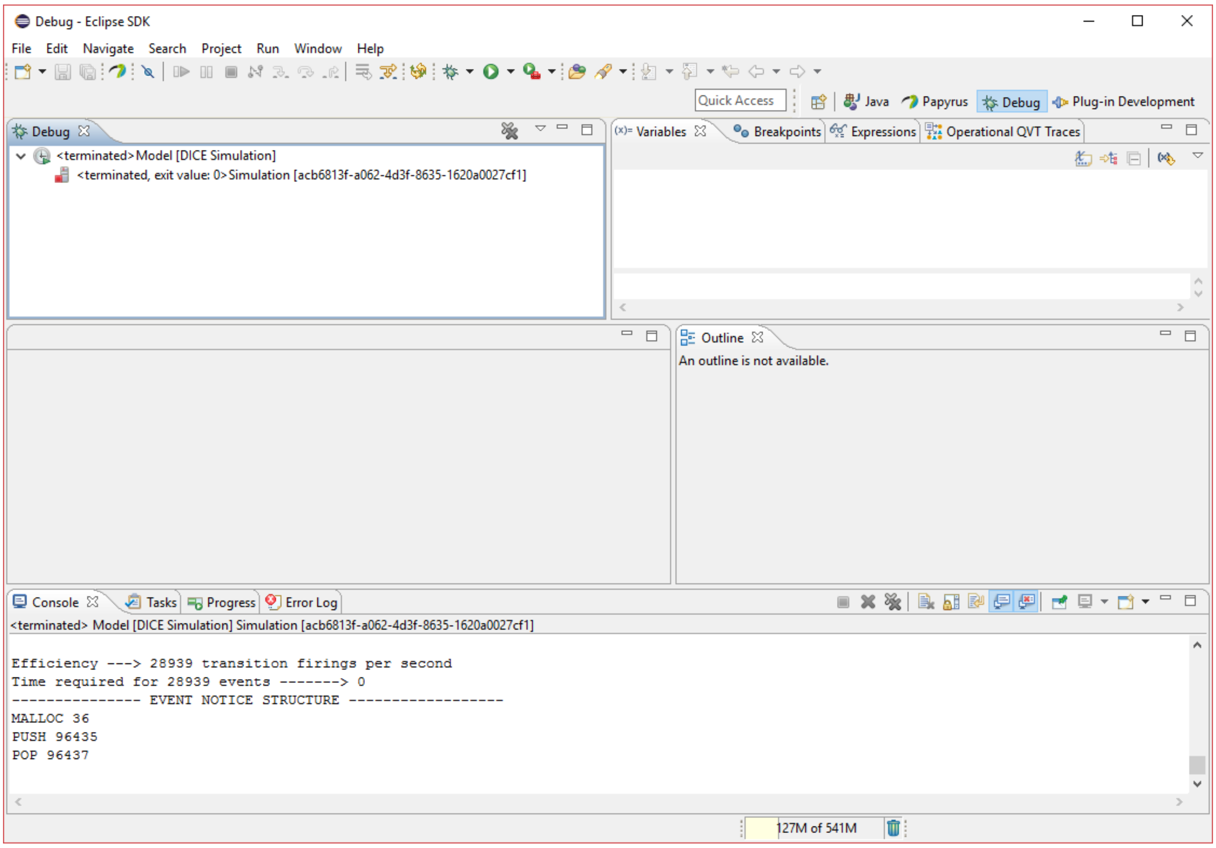

Figure 14: A finished simulation in the *Debug* perspective

Form the *Debug* view is also possible to open a window with the simulation process properties. Fig. 15 shows the properties of an example simulation. In this screenshot we can observe relevant data such as date/time in which the process was launched, process id, and raw results of the simulation.

Figure 15: Properties of a finished simulation

Figure 16 shows the normal workspace perspective once the simulation has finished. As it can be observed, in the `Test/temp` folder a set of files have been created. These are the intermediate files we previously choose to save in the workspace. These files are:

dump.pnconfig — The configuration model. This file is automatically generated by the *Configuration* module of the *Simulator GUI*.

net.pnml.xmi — An EMF representation of the PNML file that represents the Petri net corresponding to the model to be analysed. This file is automatically generated using the M2M transformation.

net.gspn.net — A *GreatSPN*-specific representation of the Petri net corresponding to the model to be analysed. This file is automatically generated using the M2T transformation.

net.gspn.def — A *GreatSPN*-specific file corresponding to the model to be analysed. This file is automatically generated using the M2T transformation.

result.txt — The result of the analysis. This file is automatically generated by *GreatSPN*.

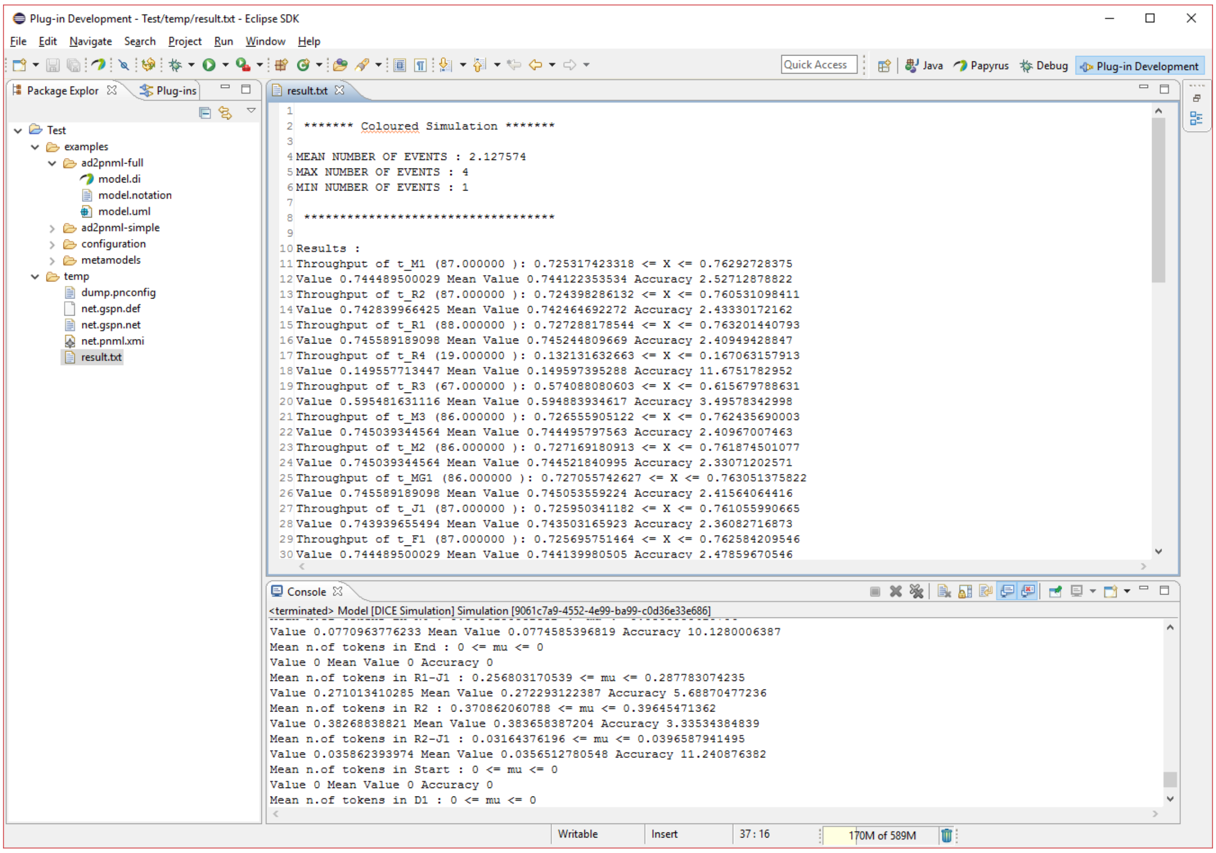

Figure 16: The workspace, showing the result of the simulation

5 Simulation Formalisms and Extensions

In order to broaden the DICE Simulation support and its community, we have also worked in the direction of extending the range of simulation formalisms that can be targeted by the DICE *Simulation Tool*. The main reason for doing this is to support multiple trade-offs between computational complexity, expressiveness of the formalism, and licensing of the external tools.

In DICE we focus in particular on two simulation formalisms:

- **Stochastic Petri Nets** (SPNs), which allow to describe a set of tokens moving across places and that require synchronized transitions. The reference tool used in DICE to analyze SPNs is *GreatSPN*, developed by a third-party (University of Turin) and released under an open source license.
- **Queueing Networks** (QNs), which allows to describe a set of jobs moving across queues and characterise their mutual contention at these queues. The reference tool used in DICE to analyze QNs is *Java Modelling Tools* (JMT) [**JMT**], maintained by IMP.

Historically, the SPN and QN formalisms have been mainstream in performance and reliability analysis. GSPNs are generally considered more flexible and better suited for formal analysis than QNs. In contrast, QNs are more efficient to analyze large-scale models and better suited to describe the effects of scheduling policies.

In order to ensure that DICE users will be in condition to run simulations on large-scale models, which are also important for the DICE *Optimization Tool*, we have therefore decided to develop a basic support for JMT, in addition to the *GreatSPN* one. Even though the primary target of the *Simulation Tool* will remain *GreatSPN*, JMT offers scalability properties that reduce the risk of incurring in computational bottlenecks on large models or across the repeated invocations invoked by the Optimization Tools. Furthermore, since JMT is developed within the consortium, it leaves us the freedom to integrate specific constructs or analyses that may not be available in GreatSPN, which we cannot change since this is developed by a third party. Below we report on some of the initial work we have done on preparing the integration of JMT with DICE.

5.1 DICE Extensions for JMT

In order to make JMT useful for DICE, we have examined its main modelling features available with the latest version (v0.9.2). We have identified in particular two main shortcomings with respect to integration with DICE:

- *QN expressiveness*. Lack of certain synchronization primitives that are important to model Big Data technologies. These include, for example, fork-join constructs with differentiated parallelism levels among job classes. These constructs are important to model certain technologies, e.g., Apache Storm, Apache Spark, and columnar databases, where different jobs can require different parallelism levels.
- *Extensibility*. JMT is lacking a plug-in mechanism for a third-party to contribute extensions to the JMT environment. Such extensions are of practical relevance to DICE. For example, we would like to develop a plug-in to support the loading of DICE-generated models into JMT. Moreover, in order to foster interest by the performance and reliability analysis community around some of the DICE modelling results, we believe that this mechanism could be used to integrate templates of models of Big Data technologies that we have defined and validated within DICE. Lastly, as technologies may change models unforeseen at this stage may need to be simulated from the DICE profile. Therefore, extensibility of the JMT tool could guarantee a better long-term sustainability for the DICE simulation capabilities, since third-parties will be able to contribute their extensions.

We describe below our work to address the two shortcomings listed above.

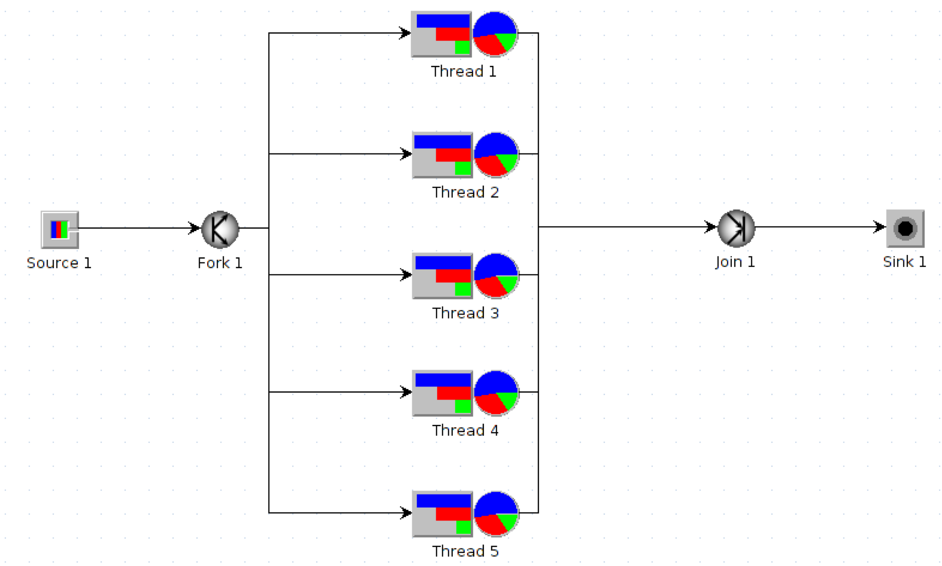

Figure 17: A JMT Queueing Network Model with a Fork and a Join operator.

5.1.1 QN Expressiveness

In order to address the lack of more expressive fork-join constructs, we have added to JMT support for probabilistic fork-join constructs. Figure 17 illustrates an example model, where a set of parallel servers is represented to describe parallel processing across a set of threads (e.g., database threads). Requests arrive from the external *Source*, and then enter the *Fork* element. In the current release of JMT, the *Fork* element would immediately split the job in a deterministic number of tasks, equal to the number of outgoing arcs from the *Fork*, and these tasks would be later reassembled at the *Join* element. This is certainly useful, but lacks the flexibility of expressing more complex fork-join behaviours present in Big Data systems.

We have therefore extended JMT to include *Variable Fork* and *Variable Join* elements, where the number of jobs forked or joined is defined probabilistically, and so is the number of outgoing links on which tasks are sent. Figure 18 illustrates the corresponding dialog window in JMT used to specify the *Variable Fork* element: the user first assigns the intended fork behaviour to a class of jobs processed by the system (step 1), then he/she assigns forking probabilities across the different output branches of the fork (step 2), and lastly expresses for each branch the probability distribution for the number of tasks to be created on that branch (steps 3/4).

As an initial proof-of-concept, we have experimentally shown in a recent paper [**CNSM**] that such elements can be useful to model real-world in-memory columnar databases. However we expect them to benefit other classes of systems such as Spark and Hadoop/MapReduce based applications.

5.1.2 Extensibility via Templates

In order to overcome the availability of a mechanism to install and manage third-party plug-ins, we have contributed to the JMT codebase an extension called *JMT Templates*. An illustration of the result is given in Figure 19. A JMT user can now download third-party extensions from a server hosted in the cloud. Upon downloading the corresponding plug-in into JMT, the user is shown in a different dialog window a list of such plug-ins, that can be activated by mouse click. The resulting behaviour of the plug-in is arbitrary as it can be coded in the JAR file that described the plug-in using Java code.

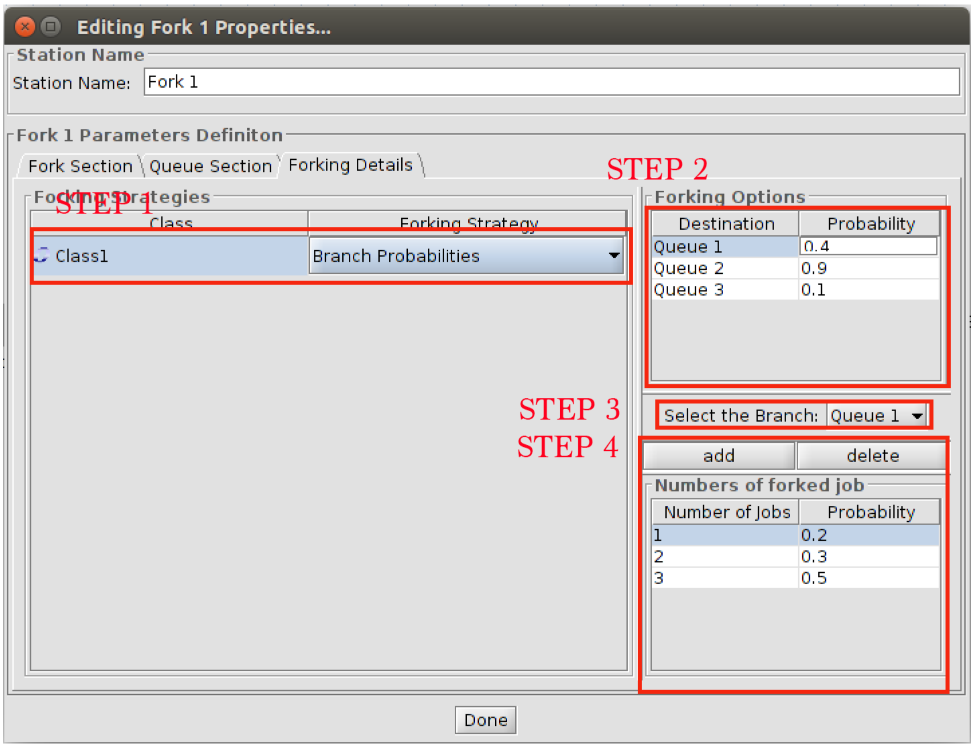

Figure 18: JMT Variable Fork-Join Extension

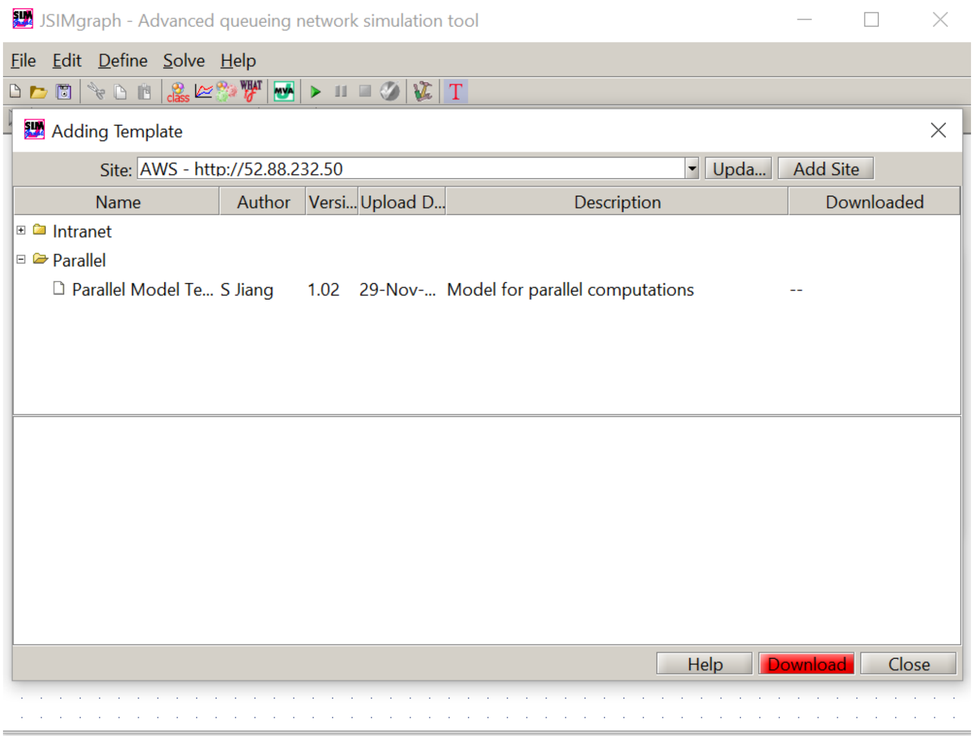

Figure 19: JMT Templates

6 Conclusions

In this document, we have presented the demonstrator of the *Simulation Tool* prototype, the main outcome of Task 3.2. At its current state, the prototype is able to cover all the steps of the simulation workflow (i.e., model, transform, simulate and retrieve results), with full integration within the DICE-IDE, and providing a user-friendly interface.

Table 1 summarizes the main achievements of this deliverable in relation to its initial objectives in terms of compliance with the initial set of requirements presented in Section 2. In the table, the labels specifying the *Level of fulfillment* could be: (i) ✗ (unsupported: the requirement is not fulfilled by the current prototype); (ii) ✔ (partially-low supported: a few of the aspects of the requirement are fulfilled by the prototype); (iii) ✔ (partially-high supported: most of the aspects of the requirement are fulfilled by the prototype); and (iv) ✔ (supported: the requirement is fulfilled by the prototype and a solution for end-users is provided).

Table 1: Level of compliance of the prototype with the initial set of requirements

Requirement	Title	Priority	Level of fulfillment
R3.1	M2M Transformation	MUST	✔
R3.2	Taking into account relevant annotations	MUST	✔
R3.4	Simulation solvers	MUST	✔
R3.5	Simulation of hosted big data services	MUST	✗
R3.6	Transparency of underlying tools	MUST	✔
R3.10	SLA specification and compliance	MUST	✔
R3.13	White/black box transparency	MUST	✗
R3IDE.1	Metric selection	MUST	✔
R3IDE.4	Loading the annotated UML model	MUST	✔
R3.3	Transformation rules	COULD	✔
R3.14	Ranged or extended what if analysis	COULD	✔
R3IDE.2	Timeout specification	SHOULD	✔
R3IDE.3	Usability	COULD	✗

As it can be seen, most of the initial (both mandatory and optional) requirements are fully or partially met. Furthermore, we have contributed extensions to the JMT tool to offer a richer simulation capability for DICE, in addition to the support for analysis based on the GreatSPN tool.

6.1 Further Work

Task T3.2 will still produce two additional deliverables in upcoming months: (i) D3.3, the *DICE simulation tools - Intermediate version* at month 24; and (ii) D3.4, the *DICE simulation tools - Final version* at month M30. For these deliverables, the following issues still need to be addressed:

- Regarding requirement R3.4, the automatic selection of the solvers is only supported by the non-UI components. Proper UIs still need to be implemented.
- Support for requirement R3.5 needs to be fully implemented.
- Regarding requirement R3.10, a proper result model together with a proper GUI to check the SLA needs to be designed and implemented.
- Regarding requirement R3.13, black box transparency will be dealt once DTSM models are addressed, currently only white box model translations are considered.
- Regarding requirement R3IDE.1, the definition of multiple metrics is supported at model level. Proper filtering and selection still needs to be implemented at the GUI level.

- Requirement R3.3 is supported via specific plug-ins and extension points only. If transformations may be selectable by end-users a better GUI is required.
- Requirement R3.14 is only supported by the core non-UI components. A new orchestrator is required, and a UI needs to be implemented.
- Regarding requirement R3IDE.2, the life-cycle of a simulation process can be completely tracked and controlled, but no specific support for timeouts has been implemented yet. This is however a minor issue.
- Support for requirement R3IDE.3 needs to be fully implemented.
- We intend to exploit the extensions described in Section 5 to produce an integration between the DICE *Simulation Tool* and JMT, since the former at the moment can target only GreatSPN.

Appendix A. The DICE-Simulation Repository

This appendix describes the *DICE-Simulation repository* [**dice:simulation:repo**]. This repository contains the following projects/plug-ins:

es.unizar.disco.core — This project contains the *Core plug-in*. The *Core plug-in* provides some utility clases for I/O, together with the shared logging capabilities.

es.unizar.disco.core.ui — This project contains the *Core UI plug-in*. The *Core UI plug-in* provides UI components that are shared across the different plug-ins contained in this repository, such as file selection dialogs.

es.unizar.disco.pnconfig — This project contains the implementation of the *Configuration Model* as an EMF plug-in.

es.unizar.disco.pnml.m2m — This project implements the M2M tranformation from UML to PNML using QVTo.

es.unizar.disco.pnextensions — This project provides some utilities to handle some extensions in PNML models. The PNML standard does not provide support for timed and stochastic petri nets. Thus, this plug-in provides the utility methods to handle this information by using the *ToolSpecifics* tags provided by the PNML standard.

es.unizar.disco.pnml.m2t — This project contains the Acceleo [**acceleo**] transformation to convert a DICE-annotated PNML file to a set GreatSPN files.

es.unizar.disco.simulation.greatspn.ssh — This project contains the OSGi component that controls a remote GreatSPN instance by using SSH commands.

es.unizar.disco.simulation — This project contains the core component that executes a simulation by orchestrating the interactions among all the previous components.

es.unizar.disco.simulation.ui — This project contains the UI contributions that allow the users to invoke a simulation within the Eclipse GUI.

es.unizar.disco.ssh — This project provides a simple extension point contribution to access a remote host by issuing the connection data using a local file.

com.hierynomus.sshj — This project contains the *sshj - SSHv2 library for Java* as an OSGi-friendly bundle. This module is required by `es.unizar.disco.simulation.greatspn.ssh` to access a remote *GreatSPN* instance using SSH/SFTP.

