DSL Based Platform for Business Process Management

Audris Kalnins, Lelde Lace, Elina Kalnina, and Agris Sostaks

Institute of Mathematics and Computer Science, University of Latvia
{Audris.Kalnnins,Lelde.Lace,Elina.Kalnina,
Agris.Sostaks}@lumii.lv

Abstract. Currently nearly all commercial and open source BPMS are based on BPMN as a process notation. In contrast, the paper proposes to build a BPMS based on a domain specific language (DSL) as a process notation – DSBPMS. In such a DSBPMS a specific business process support could be created by business analysts. A platform for creating such DSBPMS with feasible efforts is described. This platform contains a Configurator for easy creation of graphical editors for the chosen DSL and a simple mapping language for transforming processes in this DSL to a language directly executable by the execution engine of this platform. The engine includes also all typical execution support functions so no other tools are required.

Keywords: Business process management systems, Domain specific languages.

1 Introduction

Currently nearly all commercial and open source Business Process Management Systems (BPMS) are based on BPMN [1] as a process notation. The main rationale is the standardization and potential model exchange, nevertheless the process notation is only a part of a complete system definition, data model and form definitions are important as well. These BPMS aspects are not covered by BPMN, each BPMS offers its own solution there. Taking into account the complexity of the full BPMN 2.0 language it is clear that standardization does not outweigh the enhanced efforts of using BPMN for every simple process definition [2].

With the advance of domain specific languages (DSLs) in all domains of modeling and development, it is worth to revitalize also the use of DSLs for BPMS. The given paper proposes a DSL-based solution for BPMS – for domains where really a domain specific notation provides a significant gain in development speed. Since the whole development becomes domain specific, we can call the approach *DSBPMS*.

Using the DSBPMS approach it is possible to create a process definition language based on concepts and notations typical for the given domain. Then domain experts can not only read the process definitions but also create and modify them. Typical examples of such domains are insurance, healthcare, logistics etc. For example, the insurance domain could contain actions: Client Action, Broker Action, Employee Action; start event kinds: Claim Received, Risk Level Reached and domain elements used by actions: Claim, Risk etc. Similarly, processes in a healthcare institution should be based on terms understandable by doctors and healthcare personnel.

The approach is applicable also to domains where simple and flexible business processes dominate, such as internal document processing in various government institutions, for example education. There BPMN with its intricate control structures would make the notation unnecessary complicated. A simple process language based on UML activity basics would be much more suitable (see Section 4).

In this paper we propose the platform named GraDe3, by means of which the implementation of a DSBPMS even for quite a narrow domain would become feasible in the sense of efforts needed and would pay off shortly. In addition, processes in such a DSL could be easily modified to meet the goal of agile process management.

In this approach the first step is to choose a relevant domain and define an adequate process specification DSL for this domain. The platform is then used to create an advanced graphical editor set ready for building a complete process definition on the basis of the chosen domain specific process language. In addition to the editor for the process language the editor set contains editors for data model and form definitions. The next step supported by the platform is the creation of a transformation in a simple domain specific mapping language from the chosen process DSL to the language directly supported by the execution engine. In addition, this transformation defines the semantics of the created DSL in a simple and precise way. The platform contains a complete runtime support for the developed DSBPMS including user management, process execution monitoring etc., thus no other tools are required to build and execute a specific business process support in this DSBPMS. It should be noted that all steps in this development – the DSL definition, the editors for creating a concrete process management system, the transformation of the system definition to its runtime form and even the execution – are completely model-driven, with the corresponding metamodels precisely defined.

2 Related Work

There are a lot of tools, frequently named also Business Process Management Suites [3], available for the development of business process support systems. They are provided by software industry heavyweights such as IBM [4], Oracle [5], SAP [6] and others, and smaller vendors such as BizAgi [7]. In addition, a large number of open source solutions are also available – BonitaSoft [8], ProcessMaker [9] et al. Nearly all of the BPMSs use BPMN as a process modeling notation, only some use custom process languages (e.g., ProcessMaker [9]). The Gartner report 2010 on commercial BPMS [3] considers BPMN support as one of the key features in its tool evaluation. None of the popular BPMSs are based on the idea of a DSL for a process definition. Most of the suites mentioned here are very complicated to use, with a large number of service features included – they are intended to be applied in large companies with complicated business processes and with high runtime performance in mind.

One of suites most oriented to building simple systems is BizAgi [7]. Process modeling there is based on a relatively large subset of BPMN. A sort of simple E-R model is used for data modeling, there is a relatively advanced form editor and an expression language for specifying guard conditions on flows, display lists for data selection controls etc. The main difference is that the process language is fixed to BPMN, while our approach is based on a DSL having a notation adapted to the chosen kind of processes and concepts.

The open source BPMS, one of the most usable between them being BonitaSoft [8], are also nearly all based on BPMN. In addition, they typically require at least some development in an OOP language (mostly Java, including BonitaSoft).

There are very few approaches explicitly using a DSL for BPMS. One of such is the approach based on Karlsruhe's Integrated Information Management (KIM) [10], however there only a choice from existing standard process languages (BPMN, UML activity, Petri nets) is offered as a process DSL. The approach closest to ours is DSLs4BPM [11], there new graphical DSLs can be defined using the Eclipse framework, however the possible diagram structure must remain very constrained and close to the very specific PICTURE language [12] used as the base.

3 Languages and Platform for DSBPMS

3.1 General Principles of the Approach

The goal of the approach is to enable the building of a DSBPMS based on a DSL for process design with as little effort as possible. Here we want to briefly explain how the building of a DSBPMS and its usage would look like from the viewpoint of the involved stakeholders and what steps are to be done. The first step would be the choice of an appropriate domain and the conceptual design of a process DSL for this domain, with special emphasis on finding the typical kinds of custom actions. Certainly, a modeling expert is required for this task. The next task is to formalize the graphical syntax of the DSL and create a graphical editor for it using the Graphical language definition environment of the platform. A DSL developer with some skills in graphical editor building is required here. The process editor is coupled with two predefined graphical editors in the platform – for building a data model and screen forms. The next task in DSBPMS definition is to map the defined process DSL to the language directly executed by the runtime engine of the platform (the Base language, see the next section). Thus both a very simple "compiler" in the DSBPMS is built and the precise execution semantics of the DSL is defined. The skills required for this task are similar to the previous one. In some cases the execution engine has to be extended by custom libraries built using an OOP language. Now the DSBPMS is completely built and ready for use. In order to build an executable system for a business process using this DSBPMS, the process must be precisely defined in the given DSL (typically, by several related process diagrams), this task is best performed by business analysts who now all the process details. Then the process definition must be extended by the data model, form definitions for user actions in the process and expressions showing how form elements and constraints in the process are linked to the data model. This step (to be performed by a system analyst) requires some IT skills though explicit programming is not required here. After applying the mapping the system is ready to use. Fig 1 shows an overview of all these activities.

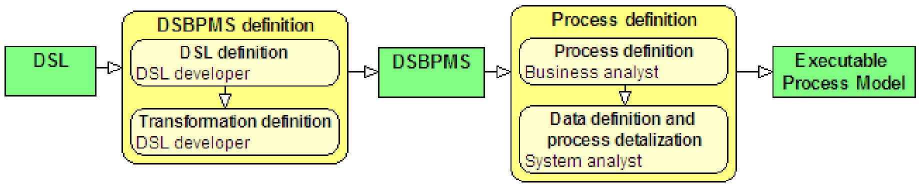

Fig. 1. Overview of the approach

3.2 Language Components for Building a Process DSL

Base Language. The proposed platform includes a process Execution engine which directly executes a simple (but functionally complete) process language, named the *Base* language in the approach. The rationale for the selection of the Base language, on the one hand, has been the simplicity of its implementation, but on the other hand, the ease of mapping the elements of a simple process definition DSL to it.

The process execution features of the Base language are chosen as a very basic subset of the UML 2.x Activity notation [13]. It includes the most used kinds of actions – general Action and CallProcessAction. Some more kinds of actions are not exactly from the UML standard, but are included because most of BPM languages (including BPMN) contain them. There are two such general subcases of action – UserAction and SystemAction. The most used kind of UserAction is ShowFormAction, and SystemAction also has several predefined sub-kinds – SendMailAction, DataAssignmentAction, CallServiceAction and a generic pattern-based AsignAllUserActions. One more important kind of elements is Custom action – CustomUserAction and CustomSystemAction. Custom actions are not directly implemented by the engine – their implementation must be supplied by the developer defining the DSL. Control structures include Start and End nodes, Decision and Merge nodes and Fork and Join Nodes, all flow control is performed explicitly by these nodes (only one flow can enter/exit an action). Only control flow edges are used, possibly with guard conditions. One more aspect is Roles and Users – each UserAction in a process has one or more Roles specified who can perform this action. The assignment of a specific user to this action is done at runtime according to the Roles.

Selection of a DSL for Process Notation. The Base language is used as a foundation for defining a specific DSL for the process notation in the given DSBPMS. It should be noted that the simplest DSL to be built is that directly coinciding with the Base language, but normally the DSL is modified to fit the chosen domain in the best way. Typically the modified elements can be directly mapped to the Base language elements or their groups. One of the goals of such mappings is to introduce derived notations for typical language constructs in the DSL. Completely new action kinds can be introduced in the DSL by mapping them to appropriate Custom actions to be implemented by the language developer.

The semantics of the defined DSL is precisely defined by its mapping to the Base language.

Data Modeling Language. Besides the process component of the DSL, each specific process management system built via the DSBPMS contains the persistent data model. The data model consists of the fixed part used directly by the execution engine and a specific data model for a given system. This specific model is to be built within a simple fixed subset of UML class diagrams. The data model is interpreted as a typical ORM image of a data base schema defining the persistent data for the given system. The fixed system runtime data can also be referenced in this custom model (using the <<system>> stereotype). Data models of existing partner systems the given system has to communicate with can be referenced as well, by the <<external>> stereotype (see a data model example in Fig. 4). It should be noted that BPMN based BPMSs (except BizAgi) typically use persistent process variables to model data (as required by the BPMN standard) which is less natural in practice (the real data are persisted in databases anyway).

Form Definition Language. Each ShowFormAction uses a specific form to be defined in a simple Form definition language. Forms in this language can contain all basic kinds of controls (textboxes, read-only fields, listboxes, checkboxes, tables as nested forms etc.). This language is also fixed in the platform. Typically one form is used in several actions, but with small modifications – some elements hidden, some made read-only etc. To support this situation in a simple way, a form with maximum details (a "main" form) can be defined and a "clone" of this form with small modifications can be easily specified for a user action. The logical structure of the form is uniquely defined by the form definition language but its style can be customized.

Expression Language. Both the process sublanguage and Form sublanguage use one fixed common element – a textual Expression language relying on the defined data model. Expressions in this language are used for binding form elements to data classes, defining the selection lists for listboxes etc. Another use of expression language is for guard conditions on control flows exiting a decision. The expression language is reused, e.g., for explicit assignment of values to process data element properties in DataAssignmentActions. Yet another use of expressions is to define parameter values for actions. Namely the last feature contributes a lot to the easy extension of functionality of the DSL. The expression language is OCL-like, however with some syntax simplification taken from OO languages, in order to support easy navigation inside the data model. Each process definition in the supported DSLs must have a base class chosen from the data model (denoted by the *self* keyword), this significantly simplifies specifying the expressions linking form and data elements. All the navigation in the Expression language is specified by using the "." symbol.

3.3 Platform Components

The proposed GraDe3 platform provides support both for the development of a DSBPMS and for its usage for building specific process management systems. Some initial ideas of such platform and mappings from the defined DSL to the execution language have been presented in [14]. In this paper the main emphasis will be on the DSBPMS development aspects.

Graphical Language Definition Environment. The graphical definition environment is based on the Transformation Based Graphical Tool Building Platform GrTP [15], which in turn is based on the TDA [16] platform. The GrTP component directly used for defining a DSL is the Configurator [17]. The definition of a diagram graphical syntax (diagram type) – its node types, edge types, their styles and their related text elements (compartments) – is also a simple diagram itself to be built in the Configurator. For compartments the relevant property dialogs can also be easily defined. Thus a completely specified graphical editor is obtained for the given diagram type.

The result of a language definition is a set of graphical editors based on TDA platform which includes the editor for the defined process DSL and the predefined editors for the data model definition and form definition. These editors constitute a workplace for developing a specific system in this DSBPMS. The generated process editor provides two views – the Business view with all data related details hidden (to be used by business analysts) and the Detailed view with all expressions visible.

Transformation/Mapping Definition Environment. A system model in the defined DSL must be converted to a model in the Base language before it can be executed. Therefore the DSL definition environment has one more component – a tool for defining a transformation (mapping) from the DSL to the Base language. In addition, this transformation defines the precise semantics of the DSL – in fact it is a very simple "compiler". Only the process sublanguage has to be mapped, the data and form sublanguages are predefined. The transformation definition is the second task for the DSL developer (see Fig. 1). We treat the DSL definition in the Configurator as a DSL metamodel, with element types corresponding to classes and compartment types to their attributes. Now a classical model transformation language could be applied. Fig. 2 shows a fragment of such a metamodel for the example DSL of this paper.

However, in most cases the transformation is so straightforward, that a simple domain specific mapping language provided in the platform is the best solution. Typically a class instance together with its direct environment in the source model (the DSL) fully determines which transformation rule is to be applied. This means that the source pattern is very simple – a class (node) instance with its attributes and incoming/outgoing edges. The corresponding target pattern normally consists of one class (node) instance in the Base language, with edges connected in a way isomorphic to the source model. In some cases an additional node instance must be added before or after the direct map target, with an edge connecting them in an obvious way (and external edges reconnected accordingly). Other elements of the source model can be transformed in a fixed way. An edge in the source maps to a target edge in the simplest way, with attributes, if any, copied. Only the predefined guard "*else*" is automatically substituted by a relevant not-based expression. A source process is mapped one-to-one to a target process and all node/edge mapping is localized inside a process.

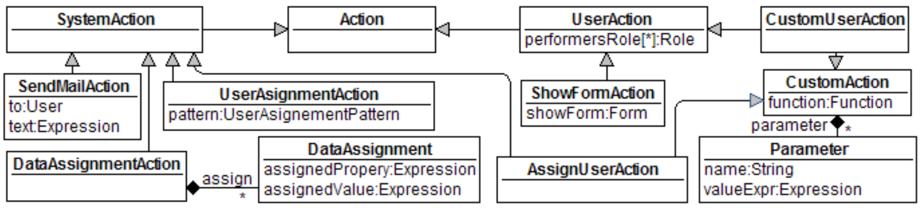

Fig. 2. Metamodel of the example DSL (fragment)

The mapping rules in the platform are defined in a tabular way, with three elements defining a mapping: the class to be mapped, the filter condition and the target class in the Base language (instance of which is to be created). If additional before/after instances must be created, their classes are specified as well. The filter condition must be specified as a Lua/lQuery expression [18]. We remind that Lua is a functional language having a collection (map) of arbitrary objects as the main data type. lQuery adds powerful expressions for filtering such collections or creating derived collections via navigating the model.

For the most typical cases predefined functions are offered, namely, the function *outLine* returning the collection of outgoing edges of a node and *inLine* returning that of incoming edges. The *size* function can be applied to a collection. The function *attr (<attribute_name>)* returns the value of the given attribute for the node (class). Expressions can be used to set the target class attributes.

The simplest mappings are for DSL elements coinciding with the Base language elements, there only the mapped class and the target class must be specified.

The two non-trivial mappings for the DSL used in this paper are described now. A simpler one using only a filter condition is that for inserting an explicit Merge node in the target when more than one flow enters an action in the source model:

```
MappedClass: Action, FilterCondition: inLine.size > 1
TargetClass: - , TargetClassBefore: Merge
```

The *Action* class is an abstract superclass therefore this rule is combined with the rule defined for each specific action class, the target class being defined in that rule.

Another more complicated mapping is used for processing the specific AssignUserAction action by mapping it to a CustomSystemAction based on the implemented function *AssignActionUser* with two parameters – a string (the action name) and a *User* class instance to be assigned. This mapping creates three class instances in the target model – a *CustomSystemAction* instance (having the chosen implementation) and two *Parameter* instances linked to the action:

```
MappedClass: AssignUserAction
TargetClass: CustomSystemAction (function=„AssignUser”)
   Parameter (name=”action”,valueExpr=attr(„actionName”))
   Parameter (name=„user”,valueExpr=attr(„userExpr”))
```

Since only one association links the used target model classes in the metamodel there is no need to specify it explicitly here.

The same mapping language facilities, especially the filter conditions and predefined functions, can be used for checking the consistency of a model in the DSL, thus a syntax checker can be easily created and only valid models need to be mapped.

Process Execution Engine. The process execution engine directly executes the Base language, with all form- and data-related actions included. It is based on the runtime metamodel implemented via a database. The engine maintains the state of all active process instances by means of tokens in a way inspired by UML activity semantics. It involves maintaining user action lists assigned to a user and automatic invoking of system actions. When an action is complete the tokens are moved, taking into account the control nodes. Due to the subset chosen, this token management is quite straightforward. All the execution is logged, in order to provide data for process monitoring.

The user management is provided via the administrator portal where users can be registered and linked to roles and process execution monitored. Regular users access the system via the user portal. There they can start a new process instance when their roles permit this. In another tab a user sees the user tasks assigned to him and ready to execute; when a task is selected the corresponding form opens.

The current version of Engine is implemented in MS.NET, with forms using the ASP.NET and data access based on Entity framework. A very valuable component of the engine is the Expression language interpreter which evaluates any valid expression over the defined data model including also the system runtime data.

4 Example – A DSL for Internal Document Processing

The example represents a simplified business trip management system in a university. Such a system should be built for the University of Latvia, but since the requirements are not finalized yet, some similar systems with available descriptions from universities in USA [19, 20] are used as a prototype for the example. The provided diagrams describe the initial part the business trip process – preparing the trip request and approving it, the whole process description would contain two more diagrams of a similar size. Though the basic path of a document in this system is quite straightforward, it contains some subtle moments related to who can actually perform the given action. In such institutions it is typical that an administrator can delegate its approval rights to another employee. The delegation rules are defined via the specific AssignUserAction included in the DSL. This is an aspect not so easily definable within the BPMN 2.0.

4.1 Description of the Example DSL

The process language of the proposed DSL reuses many of the Base language elements. However, there are some simplifications of the control structure and a new specific action is added in this DSL. The chosen DSL is well adjusted for internal document management systems; certainly, the language is slightly simplified in order to a have a complete description in the paper.

The given DSL contains the following actions – ShowFormAction, CallProcessAction, DataAssignmentAction, SendMailAction and two kinds of user assignment action – the pattern-based AssignAllUserActions taken from the Base language and one specific for this DSL – AssignUserAction. Except for the last kind, the actions are one-to-one with the Base language. The specific AssignUserAction assigns at runtime a user to the selected UserAction (by its name) in the given process instance on the basis of an expression (which must return an instance of the type *User*). It is defined by mapping it to the CustomSystemAction, a specific implementation of which is provided for this DSL (see the description of this mapping in section 3.3). According to the parameter mechanism of the Base language, both the relevant action name and the user expression are evaluated at runtime by the engine and passed as parameters to the function implementing this kind of CustomSystemAction.

The following control nodes are included in the DSL: Start, End and Decision. For simplicity we omit the concurrent flow management (Fork and Join), concurrent actions are not so frequent in internal document processing. The Merge node is substituted by the possibility to have more than one flow entering an action – certainly, all such situations are mapped to explicit Merge nodes. The mapping inserting the Merge node where required is also provided in 3.3. Thus there are only two non-trivial mappings to the Base language for this DSL, the other ones are one-to-one.

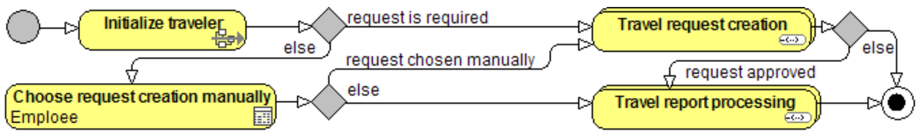

Fig. 3. Main process – University Travel System

4.2 The Example in Brief

In totality the example contains four process diagrams, one data model diagram and 5 “main” form diagrams in this system. The main process (marked as an entry point to the whole system) is the University Travel System (see Fig. 3). This process is shown in the Business view – with all data related expressions hidden. Only instances of this process can be directly created by authorized users of the system. The Role of a user who can start a new instance of this process is *Employee*. The base data class for this process is Travel (see the data model in Fig. 4), this class will be denoted by *self* in all expressions related to this process and a new instance of this class will be created at the process start. The first action to be executed in the main process is the DataAssignmentAction *Initialize Travel*. The main process invokes two subprocesses – Travel Request creation and Travel Report processing. According to the rules of University, not always the travel request has to be created before the travel (it depends on the unit where the employee works). The employee can also choose to create the request manually using the form in the ShowFormAction Choose Request creation manually.

All the expressions in the example are based on its data model, therefore we now briefly describe this model (Fig. 4). The classes without stereotypes are those built for the Travel System. The classes with the stereotype <<external>> are for some existing systems – here it is the Human Resources system. Classes with the stereotype <<system>> are those for the workflow engine runtime (only one of them is shown). We assume here that the system links each *User* instance (via the *user-emp* association) to the relevant *Employee* instance. Classes with the <<codifier>> stereotype represent instance sets for selection. Note that all classes used for the Travel System should be somehow linked to its root – the *Travel* class.

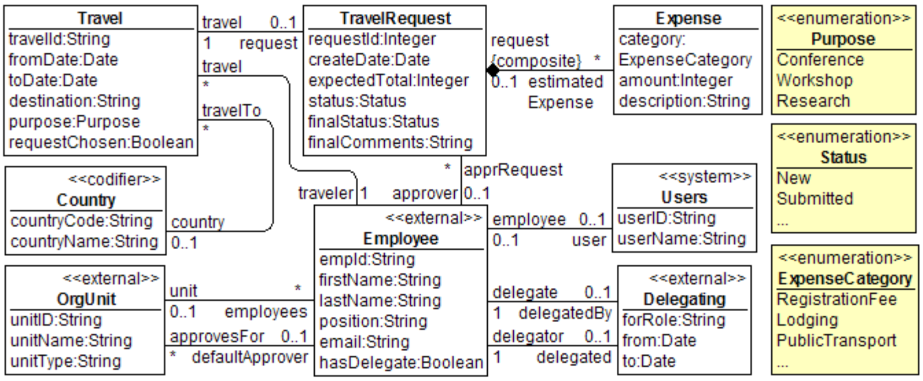

Fig. 4. Data model (the fragment used in expressions)

The process to be described in the most detailed way is the *Travel Request creation* (see Fig. 5) – it is shown in the Detailed view with all data expressions visible. In fact, the process is slightly simplified, but all used action kinds are still present. The base data class for this process is *TravelRequest*. The CallProcessAction (here the one in the main process in Fig. 3) invoking the process can specify the path in the model from the base class of the caller to the base class of the callee (here it is *self.request*).

The first action in the process is the pattern based user assignment. The pattern is "ToStarter" – the user starting the main process instance would be assigned to all actions where the filter is true – here to actions where the specified role is *Employee* (it was the starter role). The next action is the DataAssignmentAction *Initialize Request*. Two attributes of the newly created *TravelRequest* instance are set by means of built-in functions. The link to default approver (the *approver* link) is set by a more complicated expresssion (*self.travel.traveler.unit.defaultApprover*). The action *Assign Approver* assigns a user for the action *Approve Request*, by checking whether a delegate is currently set for the default approver – the setting of a delegate is done in another system (Human Resources). The SendMailAction *Notify Traveler on Final status* uses the keyword *me* for the To expression – it specifies the starter user of the whole process. The other parameter for this action is the message text. An essential ShowFormAction here is *Create Travel Request*.

To complete the example, the forms must be defined and bound to the data model. Then the defined mapping to the Base language must be run and the Travel system is ready to use. It should be reminded that no low-level programming is needed for building such a system. The example confirms the usability of the approach and the suitability of the chosen DSL for internal document processing systems.

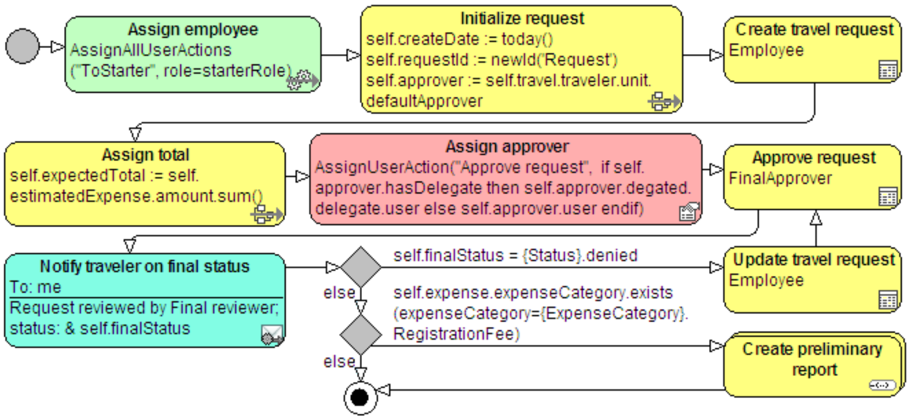
Assign employee
AssignAllUserActions ("ToStarter", role=starterRole)
Initialize request
self.createDate := today()
self.requestId := newId('Request')
self.approver := self.travel.traveler.unit.defaultApprover
Create travel request
Employee
Assign total
self.expectedTotal := self.estimatedExpense.amount.sum()
Assign approver
AssignUserAction("Approve request", if self.approver.hasDelegate then self.approver.degated.delegate.user else self.approver.user endif)
Approve request
FinalApprover
Notify traveler on final status
To: me
Request reviewed by Final reviewer; status: & self.finalStatus
self.finalStatus = {Status}.denied
else
self.expense.expenseCategory.exists (expenseCategory={ExpenseCategory}.RegistrationFee)
else
Update travel request
Employee
Create preliminary report

Fig. 5. Process diagram *Travel request creation*

5 Conclusions and Future Work

A new approach to building BPMS has been proposed in the paper. Instead of using the standard BPMN notation for business process behavior description the usage of a DSL best suited for the given process domain is recommended, thus yielding a DSBPMS for the domain. The goal of the approach is to simplify the development of a specific business process support system in such a DSBPMS so that the process development could be performed by domain experts. Certainly, all this makes sense only when the development of the DSBPMS itself can be done with relatively little effort. Therefore a new GraDe3 platform is proposed in the paper. The key components of this platform are the Configurator for easy definition of a graphical editor for the chosen DSL and a new simple mapping language for defining a transformation, by means of which process descriptions in the DSL are transformed to the language directly executable by the execution engine of the platform. An example of such a DSL is given containing two typical use cases – a domain specific control structure simplification and adding a new domain specific action kind.

The implementation of the GraDe3 platform prototype is nearly complete. The DSBPMS based on the example DSL for this paper has been built and several process examples implemented in it. Another DSL for study process management is being built and evaluated on the bachelor study management at the University of Latvia. The experiments confirm the usability of the approach for building DSLs by language designers and user-friendliness of these DSLs for business analysts. One of the key factors enabling the usability of DSLs is the included domain specific actions.

The close integration of DSL definition, transformation definition and process execution engine in the platform permits to add new features for process management. A process execution monitoring based on diagrams in the original DSL notation will be provided, as well as various queries on execution status based on such diagrams.

References

1. BPMN 2.0 specification, http://www.bpmn.org/
2. Genon, N., Heymans, P., Amyot, D.: Analysing the Cognitive Effectiveness of the BPMN 2.0 Visual Notation. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 377–396. Springer, Heidelberg (2011)
3. Sinur, J., Hill, J.: Magic Quadrant for Business Process Management Suites. In: Gartner RAS Core Research Note G00205212 (2010), http://www.gartner.com/id=1453527
4. IBM Business Process Manager, v 8.5, http://www-03.ibm.com/software/products/us/en/business-process-manager-family
5. Oracle Business Process Management Suite 11g, http://www.oracle.com/us/technologies/bpm/suite/overview/inex.html
6. SAP NetWeaver BPM, http://scn.sap.com/community/bpm
7. Bizagi BPM suite 10.1, http://www.bizagi.com/index.php
8. BonitaSoft - Bonita Open Solution, Open Source BPM, http://www.bonitasoft.com
9. ProcessMaker Workflow management and BPM, http://www.processmaker.com
10. Freudenstein, P.: Web Engineering for Workflow-based Applications: Models, Systems and Methodologies. KIT Scientific Publishing, Karlsruhe (2009)
11. Heitkötter, H.: A Framework for Creating Domain-specific Process Modeling Languages. In: Proceedings of the ICSOFT 2012, pp. 127–136 (2012)
12. Becker, J., Pfeiffer, D., Räckers, M.: Domain specific process modelling in public administrations–the PICTURE-approach. In: Wimmer, M.A., Scholl, J., Grönlund, Å. (eds.) EGOV 2007. LNCS, vol. 4656, pp. 68–79. Springer, Heidelberg (2007)
13. UML specification v.2.4.1, http://www.omg.org/spec/UML
14. Lace, L., Liepiņš, R., Rencis, E.: Architecture and Language for Semantic Reduction of Domain-Specific Models in BPMS. In: Aseeva, N., Babkin, E., Kozyrev, O. (eds.) BIR 2012. LNBIP, vol. 128, pp. 70–84. Springer, Heidelberg (2012)
15. Barzdins, J., Zarins, A., Cerans, K., Rencis, E., et al.: GrTP: Transformation Based Graphical Tool Building Platform. In: Proc. of MDDAUI 2007 Workshop of MODELS 2007, Nashville, Tennessee, USA, CEUR Workshop Proceedings, vol. 297 (2007), http://ceur-ws.org
16. Kozlovics, S., Barzdins, J.: The Transformation-Driven Architecture for interactive systems. In: Automatic Control and Computer Sciences, vol. 47(1/2013), pp. 28–37. Allerton Press, Inc (2013)
17. Sprogis, A.: The Configurator in DSL Tool Building. In: Scientific Papers, vol. 756, pp. 173–192. University of Latvia (2010)
18. Liepiņš, R.: Library for model querying: IQuery. In: Proceedings of the 12th Workshop on OCL and Textual Modelling (OCL 2012), pp. 31–36. ACM, New York (2012)
19. Welcome to e-Expense Travel & Business Expense System. A User Guide, Tufts University, http://finance.tufts.edu/accpay/files/eExpenseGuide.pdf
20. The Ohio State University, eTravel ASSIST, https://assist-erp.osu.edu/assistTravel/index.htm