A Build Server for Model-Driven Engineering

Henrik Steudel, Regina Hebig, and Holger Giese
Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany
{forename.surname}@hpi.uni-potsdam.de

ABSTRACT

Model-driven engineering (MDE) is more and more used in collaborative settings. Therefore, the usage of build servers to gain early integration of different system parts is desirable. Current build server technologies only consider fully automated operations. Further, the availability of all described artifacts is necessary for a correct run. However, in MDE manual activities can occur in between automated operations, which also prevents that all necessary artifacts are already available when the build starts. Therefore, state of the art build techniques are not sufficient to support MDE development. In this paper, we present a build server prototype which is designed to fit the needs of development with multiple paradigms and languages.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

Keywords

Model-driven engineering, build server, integration process

1. INTRODUCTION

Nowadays, model-driven engineering is used to tackle the increasing complexity of software. Models and code are employed to represent different levels of abstraction or different aspects of a system following different paradigms. Model transformations or other model operations further automate some of the required development steps and permit to combine the multi-paradigm model into a consistent whole.

In code-centric development build servers provide early feedback on the integration process. They execute operations to transform development artifacts to an executable or deployable product. Automatic verification is employed to check the product for specified semantic properties.

In model-driven engineering (MDE) multiple languages are used to create a multi-paradigm model of the developed system. Thereby, models are combined to describe different system aspects or are used on different levels of abstraction. Therefore, content is propagated via transformations between artifacts. This can require the mixed application of manual and automated steps during implementation. Examples are described in several studies [6, 8, 9, 14, 11]. Such *MDE settings* are used in multiple projects [8].

Classical build approaches for code-centric development include only fully automated steps, e.g. compiling source code or creating a deployable product. These approaches only work if all artifacts described in the build script are provided. Further, in classical build servers the outcome of verification and validation (V&V) activities has either no influence or leads to a complete break of the build. Thus, builds are not executed partially.

However, in MDE also manual operations have to be applied in between different automated operations. A build approach needs to be able to handle these manual operations. Due to manual operations, it is further possible that artifacts are missing during build. Finally, several V&V activities might work on results of preceding manual activities. It is desirable to use such verifications as early as possible to prevent the application of other expensive, possibly manual, activities. Thus, an integration of V&V activities within the build script is necessary.

Subsuming, current build approaches do not support MDE needs. Languages for build scripts do not allow capturing manual activities or integrated V&V activities. In addition, there are no explicit mechanisms to handle manual involvement, deal with the fact that manual activities require much more time than automated ones, or decide whether manually contributed artifacts are outdated.

In this paper, we present a build server prototype that provides support for MDE specific requirements. We introduce an integration meta model that allows specifying build scripts with manual and V&V operations. Further, we developed concepts and a build process that can handle situations with missing artifacts or failing integrated verifications.

The paper is structured as follows. In the next section we discuss MDE specific requirements on a build server in detail. Section 3 introduces the build server with the integration meta model for the definition of build scripts. Further the build process is explained and we discuss how the requirements introduced in Section 2 are fulfilled. In Section 4 we discuss related work and conclude in Section 5.

2. REQUIREMENTS

The above described property of MDE, i.e. the mixture of manual and automated activities, leads to special needs for a build server. In the following, we introduce these needs and discuss that some of them might be also relevant for non-MDE projects. However, we will show that the mixture of manual and automated activities in MDE makes these needs much more pressing.

Handling Manual Operations. The usage of outdated artifacts during the build process should be prevented. Therefore, a build server has to be able to identify, whether manual operations have to be executed to update an artifact. *Capturing manual activities* in the build script is a precondition for such an analysis. Manual activities need much more time than automated ones, since they are constrained by the availability of human resources and are often not trivial. Therefore, a build server for MDE needs to be able to *deal with missing artifacts* that will result from manual activities. Further, a build server needs to *provide resulting artifacts* of a build, so that they can be used during manual activities. Finally, an approach has to *notify users* about the need to execute manual steps. To prevent users from working with outdated artifacts, they should be notified when input artifacts for manual activities are changed (e.g. by automated operations executed during a build). However, in terms of usability, the notifications should only be sent if necessary and not automatically with each build.

Integrated V&V Operations. For both MDE- and non-MDE projects it is, especially for troubleshooting, useful to identify errors before they are propagated via transformations to other artifacts. However, there is a pressing need for this ability in MDE, when potentially erroneous artifacts might be used for performing expensive manual activities. Further, checks may be necessary as preconditions for an automated activity to ensure correct behavior. However, verification that is performed at the start of an MDE build cannot capture all errors, when design and implementation decisions are made in later manual steps. To enable tradeoffs between these needs, we identify the requirement that a build approach should allow to *specify and execute verification and validation operations* on artifacts produced and consumed in the build script.

Lightweight Build Script and Partial Builds. When a build has to handles manual operations, as in MDE, it is important to discuss whether users using the build server are restricted in their behavior. A build server might enforce a specific sequence of operations or might not be able to handle spontaneous iterations of one or more manual activities. A build script language and execution semantic should support the definition of **lightweight build scripts** that allow maximum freedom of the execution order. The goal is a reduction to technically necessary sequences, as they are determined by production and consumption of artifacts.

Similarly, a build language should not require the user to define in each build script appropriate alternative behavior for handling failing verifications and situations with missing artifacts (e.g. due to necessary manual activities). Such behavior should be built-in into the execution semantic of the build script. A build server should not simply abort a build with the first occurring problem, skipping operations that are ready to be executed. Instead, operations should be executed whenever possible, since in MDE operation products can be prerequisite for further manual operations that follow on automated operations. A controlled **partial build**, as far as it is possible with the given and successfully verified artifacts, should be performed. Partial builds can also be interesting for builds in non-MDE settings, e.g. produced partial results might be used for troubleshooting.

3. A BUILD SERVER FOR MDE

The build server's main task is the execution of a build script on artifacts (i.e. models or code) that are stored within a version control system (VCS). The artifacts created during the build are committed to the VCS, where they can be checked out by users to be used for manual activities. Foundation for handling described MDE needs is the design of build script language and build process. The build script is represented in form of an *integration model*, that specifies an object flow of artifact roles between manual and automated activities. Artifact roles allow to formulate the build script independent of a specific project and serve as placeholders for artifacts. During the project artifacts committed to the VCS (either by users or as result of a build execution) are mapped to artifact roles. Additionally, verifications can be specified in the build script between artifacts to prevent the usage of incorrect artifacts in following operations, or between artifacts and operations as precondition for the application of operations. While automated activities are directly executed during the build whenever possible, manual activities cannot be executed during a build. Here the build server provides a messaging concept, that is utilized to provide users with information in case manual activities have to be executed. When results are committed to the VCS and mapped to the appropriate artifact roles, the build server can use them in the following builds. We will introduce the architecture of the build server, followed by the introduction of the integration meta model that is used for the specification of build scripts, and the description of the build process.

3.1 System Design

This section outlines relevant build server components of our prototype as depicted in Figure 1. The Build Manager is the central component of the build server. It integrates all functionality needed for the execution of builds. Therefore it provides access to the VCS and third-party technologies. Additionally, the Build Manager contains the current build script instance. The communication components establish a communication protocol to enable client-server messaging. This messaging is embedded in the development environment of the client. The build server sends notifications to clients and receives artifact mappings from clients. Technology Adapter plugins adapt external technologies for actual execution of verifications and model operations. Each adapter claims responsibility for a technology which can be referenced from build scripts. The set of supported technologies can be extended by supplying additional adapter plugins. Build Jobs are part of the Build Manager and represent specific build execution instances. Every job operates on the *build model*, a copy of the current build script from Build Manager. Also it uses the Build Manager to access the VCS and technology adapters. Basic job configuration requires an assigned build script and a build trigger. Triggers may either specify periodical or on-demand execution, e.g. a build is requested by a user.

3.2 Integration Model

The integration meta model (shown in Figure 3) consists of three parts. The main part (illustrated with black lettering) allows the specification of the build script. This part of the integration model is independent of a specific project and can be reused for equal MDE settings. The second part of the integration meta model (illustrated with blue lettering)

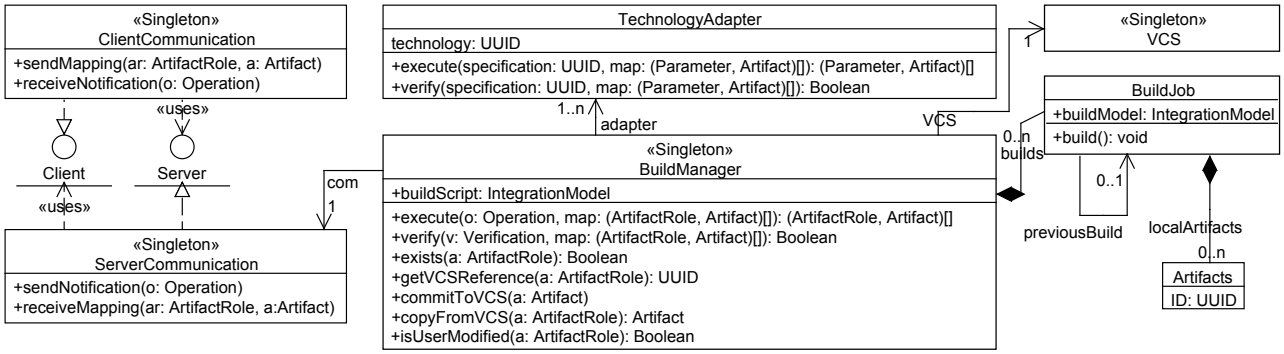

Figure 1: Design of the build server

allows to specify the mapping of project-specific artifacts to project-independent artifact roles in the build script part. Finally, the integration meta model allows 'process annotations' (illustrated with gray lettering) that are specific for each build job and only used within the build model copies. To illustrate the meta model we use the example of a factory production modeling process taken from [11]. Starting from initial structural (SDL block diagrams) and behavioral specifications (Storycharts) the process combines automated and manual model operations to derive a compile-ready code base. Figure 2 shows a possible integration model for that process, which is augmented with sample verifications.

3.2.1 Build Script

The meta model basically contains three types of elements: artifact roles, verifications, and operations. Initial artifacts, like the example's blockdiagram and storycharts, constitute the entry point for build processing. An operation holds two UUIDs to reference its specification (i.e. the implementation) and the technology, which can be used to execute this operation (e.g. ATL modules). Further an operation can be marked as manual, e.g. operation 'extended by custom elements'. In the example's illustration automated operations are decorated with a 'gears' symbol, whereas manual operations bear a 'persona' symbol. Finally, an operation holds sets of input, output, or in-output parameters, each referencing an artifact role. Thus, the set of produced and consumed artifacts can be specified for an operation. In-output parameters can be used to indicate that an operation respects a former output during reapplication.

The integration model provides a set of verification types. Similar to operations, a verification holds two UUIDs to reference its specification (e.g. a ruleset) and the technology, which is employed to execute the verification. Further, a verification references all used artifacts. There are two possible targets for a fail of a verification. Either artifacts are not used further during the build or an activity is not used during the build. In addition, a verification can either focus on the correctness of artifacts (concerning their syntax, semantic, and compatibility) or on the correct execution of an activity. Verifications that have the consequence, that a subset of the used artifacts (referenced as impacted artifacts) is not employed as input for operations during the build, are modeled as artifact verifications. Verifications that focus on the correctness of artifacts and target the usage of artifacts during the build are horizontal consistency checks and solo artifact checks. Horizontal consistency is defined on a set of artifacts that mutually do not form a source-result relationship [13], e.g. 'chart per process' checks whether there exists a story chart for each process in the block diagram. Solo artifact checks are similar to horizontal consistency checks, but only defined on a single artifact, e.g. 'activities specified' in Figure 2 that checks if all activities contain a behavior specification. Verifications that focus on the correctness of artifacts and target an activity are postconditions. Artifacts used in postconditions are result of the same operation, e.g. 'methods specified' checks whether code fragments were assigned to appropriate methods. Since an operation is already executed, when a postcondition is performed, the consequence of a fail is that the checked artifacts are not used further. In these three cases all used artifacts are also impacted artifacts, e.g. if 'Sourcecode' fails the 'methods specified' check, it will not be used further. Verifications that focus on the correct execution of an activity and target the usage of artifacts during the build are vertical consistency checks. Vertical consistency checks whether one impacted artifact a is consistent with one or more *ancestor artifacts*, i.e. artifacts that are direct or indirect input for operations creating a. 'Class per process' checks, whether for each process in 'SDL block diagram' a class in the 'extended class diagram' exists. Only then 'extended class diagram' can be used to execute operation 'generate class stubs'. Finally, verifications that focus on the correct execution of an activity and target an activity are preconditions. Used artifacts of a precondition are input of the impacted operation. If a precondition fails, the impacted operation will not be executed during the build. For example, the precondition 'match' in Figure 2 checks, whether the 'state table implementations' can be matched to corresponding 'class stubs'. Only then the following operation is executed.

3.2.2 Artifact Mapping

The model's second purpose covers mappings between the model's logical elements and their representation in the repository. Artifact roles are assigned to development artifacts via *artifact* property. A mapping is not necessarily given for each artifact role, as artifacts might not yet be created.

3.2.3 Processing annotations

Processing annotations are determined at build time and gather information about reusability of previous build results as well as an execution status. Artifact role defines properties *reusable* and *reused* if the artifact is eligible for reuse, respectively if it was actually reused during the build. The property *user-modified* remarks if a new version of the

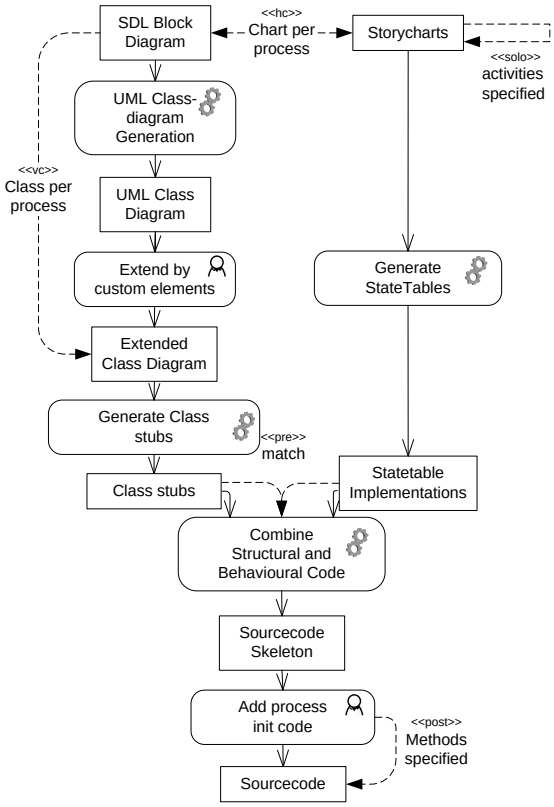

Figure 2: Factory Example

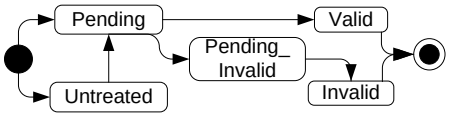

Figure 4: Verification States of an Artifact

artifact was committed to the VCS since the previous build started (excluding the previous build's commits). Operations and verifications define an *executed* property which is set after execution to avoid redundant runs. The outcome of a verification is stored in *result* property.

The states of an artifact role (see Figure 4) reflect the current verification status. The default state UNTREATED indicates that an artifact role is either not mapped to an artifact within the build or it is not yet decided, whether the artifact will be (re-)used during the build. Artifact roles that are in state PENDING or PENDING_INVALID are subject to the application of verifications. In case of PENDING_INVALID at least one verification that influences the validity of the artifact role already failed. Nonetheless, remaining verifications on a PENDING_INVALID artifact still have to be executed, since they can affect other artifacts, too. For artifact roles that are in state INVALID no further validations have to be applied and at least one verification failed that influences the validity of the artifact role. Finally, the state VALID indicates that the artifact role can be used as input for operations during the build. Similar to artifact roles, operations pass through the same states (except UNTREATED) to determine whether they can be applied during the build.

3.3 Build Process

In this section we present the overall build processing, followed by explanations regarding incremental builds and manual activity handling. The build process is started by a configured build trigger. At this time a copy of the build script in Build Manager is supplied to the build job for interpretation and storage of processing annotations (build model).

3.3.1 Overall Build Process

The build process starts with an initialization phase, proceeds to actual build execution phase and afterwards closes the build. Build initialization first creates a local build environment where all current versions of mapped artifacts are copied to from VCS. Artifact roles now point to artifacts in the local pool instead of the VCS. All artifacts that are determined for potential regeneration are locked in the VCS. This prevents that a user can commit changes to artifacts that might be overwritten by the build job's commit. Further, states of initial artifact roles are set to PENDING. The build execution itself loops through the following sequence of actions. First, solo artifact checks and horizontal consistency checks are performed on PENDING artifact roles. These verifications might require not yet available artifacts and, therefore, must be postponed. However, if all verifications defined on an artifact could be performed successfully, the artifact is deemed VALID. Valid artifacts in turn enable execution of model operations. Thus, the next build step inspects operations for valid input artifacts and performs precondition verifications. Thereby, any in-output parameter's state is not relevant. On successful evaluation the operation is eligible for application. Before the operation execution is triggered, we check whether previously created output artifacts can be reused. If this is not the case, the operation has to be executed. Therefore, automated operations are delegated to the corresponding technology adapter. In contrast, for manual operations a notification is sent to the users. Either way it is assumed that generated artifacts are mapped to output artifact roles by the executor. The states of the output artifacts are set to PENDING. Afterwards, postconditions and vertical consistency checks are performed. At this point the loop starts over. New artifacts may enable further verifications and subsequent model operations. The loop is exited when last iteration did not contribute any output artifact to the pool of artifacts. This is either the case if all defined operations actually have been executed (**complete build**). Alternatively, remaining operations lack valid input artifacts or failed a precondition check (**partial build**). Subsuming, the build process has carried out all eligible verifications and model operations. It terminates gracefully despite possible incomplete artifact mappings or verification failures. After termination every generated artifact is made available to the users. Hence, on build closure these artifacts are committed to VCS. Finally, the build job is prepared to be used for reusability determination in the next build job. Therefore, relevant artifact mappings in the current build model are adapted to reference the artifacts now located in the VCS. For artifacts that are committed the first time, a mapping is added to the Build Manager's build script.

3.3.2 Incremental Builds and Reusability

The build process pursues an incremental approach to minimize the amount of required verification and operation executions. We evaluate whether results from a previous ex-

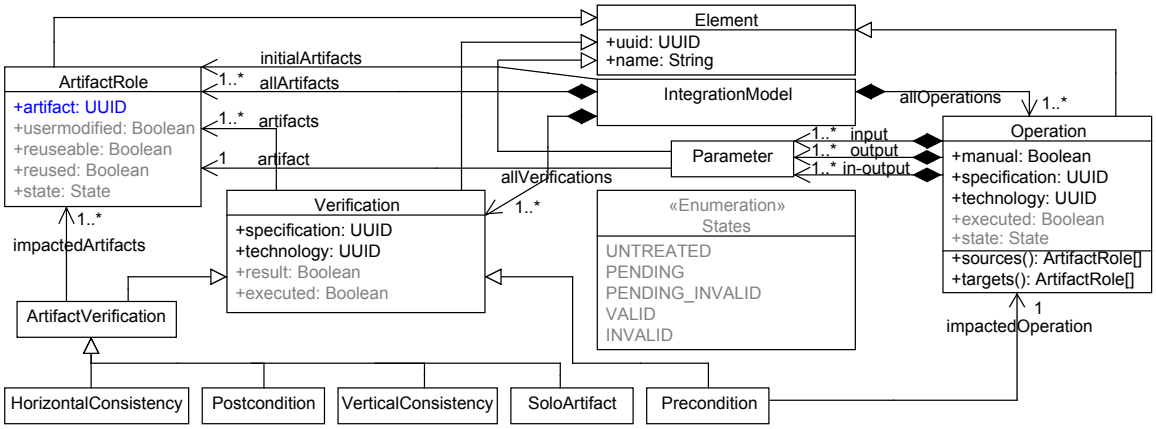

Figure 3: Meta model of the integration model

ecution of an operation can be reused, instead of executing this operation again. This is possible if all input artifacts of the operation are already reused and output artifacts are mapped in the previous build. During initialization the build process detects artifacts that are modified by users since the last build started. Thereby, it is considered when artifacts were updated in the repository. For these artifacts the corresponding artifact roles are marked as *user-modified*. In case an input artifact or one of its ancestor artifacts is user-modified the operation's results cannot be reused. If a verification exclusively covers reused artifacts, it is possible to reuse the verification's result from previous build.

Above is described how information from the previous build can be employed for reuse. Therefore, the adaptation of artifact mappings during build closure have to be done in a specific manner. There are two kinds of artifact roles for which mappings to corresponding artifacts in VCS are set. First, there are the artifact roles that are not in the state UNTREATED. For example, these are artifacts that were reused or successfully generated and subsequently committed. Second, there are artifacts that remained in state UNTREATED, but are marked as *reusable*. The property reusable is determined statically at the start of a build and indicates that no ancestor artifacts have been user-modified. For reasons like failing horizontal checks the partial build has stopped before the producing operation was considered for execution. Nevertheless, the artifact is still eligible for reuse. However, there are artifact roles which must not reference the artifact in VCS any longer. These artifacts also remained in state UNTREATED at build closure but are not marked as *reusable*. So, ancestor artifacts were user-modified but the current artifact is not yet updated based on this new input.

3.3.3 Manual Activity Handling

As previously noted, a manual operation has to be performed whenever its input artifacts are modified. The build job dispatches notifications to users, containing the task and involved source artifacts. As the current build job has not received any results yet, it simply treats the output artifact roles as missing artifacts. Any notified user can carry out the operation and commit the resulting - new or updated - artifacts back to VCS. In case of new artifacts the user is also required to create the mappings to corresponding target artifact roles. Thus, these artifacts are introduced as user-modified artifacts and can be used in subsequent build jobs. The build server sends notifications whenever modified input for manual operations is detected. On that basis we can make the reasonable assumption that users consider latest input artifacts when committing resulting artifacts. This also applies in the case that source and target artifacts of the same operation are user-modified.

3.4 Discussion

Following, we discuss how the above presented build approach fulfills the requirements listed in Section 2. The integration model is designed to specify usage of manual and automated operations. Missing artifacts don't prevent an execution of the build, due to the partial build concept. Thus, the user is always provided with all artifacts that can be built successfully. Manual operations are handled by sending notifications to the user. Thereby, he is always notified about the current version of the input artifacts, but gets no obtrusive repeated requested if nothing changed. The integration meta model further allows to specify the applications of integrated V&V operations. We associate different verification types with a built-in semantic for consequences in case of failures. This enables the build process to decide which operations can be meaningfully executed. In consequence users are not asked to perform manual activities if verifications on input artifacts fail. Finally, the integration meta model is designed for build scripts to be as lightweight as possible. We implement a solution where no control flow but only the object-flow is defined, i.e. it is specified which artifacts are consumed and produced during operations. This declarative way of definition allows the build approach to only minimally restrict the order of manual operations. The build process makes no assumptions about when or in which order updated versions of artifacts can occur in the VCS. Therefore, the user is free to perform manual activities in any order and the build server can deal with spontaneous iterations.

4. RELATED WORK

The actual expressiveness of build servers like Hudson[1] or Jenkins[2] bases on the used build scripts. Languages for specifying build behavior are Make [5], ANT [1] and Modeling Workflow Engine 2 (MWE2) [2]. Make and ANT are rather lightweight. These technologies provide no support for integrated V&V operations, like preconditions for the execution of compilation steps. Maven 2 [7] scripts define combinations

[1]http://hudson-ci.org/
[2]http://jenkins-ci.org/

of plugins supporting different technologies and are bound to a build lifecycle. The default lifecycle mixes operation execution and verification phases. Although they are very popular, none of these languages was designed to explicitly deal with manual operations. Finally, the above mentioned build languages support no partial builds. ANT processes defined operations independent from verification results or missing artifacts. In Make the build fails completely in case of missing artifacts. MWE may be used to explicitly define alternative behavior on failing verifications. This leads to less lightweight build scripts.

Languages like MoScript [10], MoTCof [15], or Epsilon [12], provide lightweight concepts for explicit composition and automated execution of model operations. However, support for integrated V&V operations is rather rare. Epsilon [12] is an example where verifications can be explicitly modeled as part of the composition. None of these approaches provides support for handling of manual operations. Also partial execution is not supported, e.g. a verification that fails in Epsilon leads to full abort of the execution.

Software processes by nature have to deal with manual tasks. In context of MDE, some attempts aim at providing support for automated execution of software processes. An example is the UML4SPM language [4, 3], which allows to model manual activities as part of the executed process. Communication and exchange of results between activities is supported. Unfortunately, it is not discussed in [4, 3] how the system reacts to long waiting times during the execution of manual activities. Since UML4SPM was not created with the intention to support builds, it is not surprising that the authors describe no concept of a partial execution. While verifications in the process can be modeled as activities, consequences of failing verifications have to be modeled explicitly as alternative process paths. As a process language UML4SPM can be used to recreate partial builds, which leads to more complex processes. Thus, UML4SPM provides no lightweight language for build scripts. Subsuming, combining automation and manual involvement is only in focus of UML4SPM [4]. Partial builds and integrated V&V operations have no built-in support in the discussed approaches.

5. CONCLUSION AND FUTURE WORK

In this paper, we discussed special requirements on build servers for the treatment of MDE approaches. Thereby, it was identified that a build server requires support for manual operations, integrated V&V operations, and partial build runs. We presented an integration meta model that allows to create build scripts for MDE and introduced a prototypical build server. Further, we discussed how our approach fulfills the identified requirements on a build server for MDE.

In future work we will allow also manual verifications. Further, the build server might be used for statistic analysis in case the same integration model is reused in several projects. For example, additional verification might be formulated for artifact roles that turn out to be subject to frequent rework.

Acknowledgments

We thank Thomas Beyhl for his support during the implementation of the prototype.

6. REFERENCES

[1] S. Bailliez, N. Barozzi, and J. Bergeron. *Apache Ant User Manual*. The Apache Software Foundation, 2003.

[2] H. Behrens et al. *Xtext User Guide*, chapter 6. 2010.

[3] R. Bendraou, M. Gervais, and X. Blanc. UML4SPM: A UML2.0-Based Metamodel for Software Process Modelling. In L. Briand and C. Williams, editors, *Model Driven Engineering Languages and Systems*, volume 3713 of *LNCS*, pages 17–38. Springer, 2005.

[4] R. Bendraou, J.-M. Jézéquel, and F. Fleurey. Achieving process modeling and execution through the combination of aspect and model-driven engineering approaches. *Journal of Software Maintenance and Evolution: Research and Practice*, 2010.

[5] S. I. Feldman. Make – a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[6] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In G. Engels, B. Opdyke, D. Schmidt, and F. Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 482–497. Springer, 2007.

[7] T. A. S. Foundation. *Apache Maven User Guide*. The Apache Software Foundation, 2011.

[8] R. Hebig and H. Giese. MDE Settings in SAP. A Descriptive Field Study. Technical report, Hasso-Plattner Institut, University of Potsdam, 2012.

[9] J. Johannes and U. Assmann. Concern-Based (de)composition of Model-Driven Software Development Processes. In D. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems*, LNCS. Springer, 2010.

[10] W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot. MoScript: A DSL for querying and manipulating model repositories. In *Software Language Engineering (SLE)*, Braga, Portugal, 2011.

[11] H. J. Köhler, U. A. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the* $22^n d$ *International Conference on Software Engineering (ICSE), Limerick, Ireland*, pages 241–251. ACM Press, 2000.

[12] D. Kolovos, R. Paige, and F. Polack. A Framework for Composing Modular and Interoperable Model Management Tasks. In *MDTPI workshop, EC-MDA, Berlin, Germany*, June 2008.

[13] J. Küster and G. Engels. Consistency Management within Model-Based Object-Oriented Development of Components. In *Proceedings of the conference on Formal Methods for Components and Objects (FMCO 2003), Leiden, The Netherlands*, pages 157–176. Springer, October 2004.

[14] B. Roussev and J. Wu. Transforming use case models to class models and OCL-specifications. *Int. J. Comput. Appl.*, 29(1):59–69, January 2007.

[15] A. Seibel, R. Hebig, S. Neumann, and H. Giese. A Dedicated Language for Context Composition and Execution of True Black-Box Model Transformations. In *4th International Conference on Software Language Engineering (SLE 2011) , Braga, Portugal*, July 2011.