Artifact-based functional comparison of software processes

Rodion M. Podorozhny
UT Advanced Research In Software Engineering (UT ARISE)
University of Texas
Austin, Texas 78712
podorozh@mail.utexas.edu

Dewayne E. Perry
UT Advanced Research In Software Engineering (UT ARISE)
University of Texas
Austin, Texas 78712
perry@ece.utexas.edu

Leon J. Osterweil
Laboratory for Advanced Software Engineering (LASER)
University of Massachusetts
Amherst MA 01003
ljo@cs.umass.edu

ABSTRACT

Sound methods of analysis and comparison of software processes [1] are crucial for such tasks as process understanding, process correctness verification, evolution management, process classification, process improvement, choosing the appropriate process for a certain project.

The aim of this work is to lay the foundations for a systematic and rigorous comparison of processes by establishing fixed methods and conceptual frameworks that are able to assure that comparison efforts will yield predictable, reproducible results.

The comparison process presented here assumes that the comparison will be done relative to a fixed standard feature classification schemas for the processes used, and with the use of a fixed formalism for modeling the processes. The aspect of the process described in this paper is focused on functional comparison of processes according to the predefined comparison topics. The process assumes that the compared processes are functionally similar at the top level of abstraction. The suggested comparison process is based on the comparison of artifacts [2] produced by the processes when given the same input. This fixed feature classification schema for artifact elements is used as a point of reference for the comparison and it enables one to avoid making conclusions based on interpretation of process activities' names.

The paper describes our comparison process and its application comparing two versions of a logistics software process from the telecommunication domain.

[1]A software process is thought of as a way of producing products or delivering services specified in software and requiring human involvement

[2]This paper uses the term "Artifact" to denote a data structure of a software process.

Keywords

Software Process, Comparison

1. INTRODUCTION

This work presents a novel approach for comparing software processes that enables one to significantly increase the objectivity and repeatability of comparisons. To our knowledge, this is the first attempt at a partially automated comparison of software processes based on the artifacts they produce. While our work focuses on the application of the comparison process to software process comparison, it is more general. It is also applicable in domains other than software process, such as data-based comparison of software applications for evaluation of continuous program optimization techniques ([8]).

It is our belief that certain tasks (e.g. software development) are very unlikely to be completely automated in the foreseeable future if ever. Thus there will be a need for software process systems with human involvement in their execution. We believe that the operation of such systems can be properly described and analyzed with the use of the concept of a software process as introduced in [10].

One of the hallmarks of a mature scientific or engineering discipline is its ability to support the comparison and evaluation of the artifacts with which it deals. Systematic comparisons rest upon classification. Thus we believe that the establishment of a discipline of process engineering requires the development of techniques and structures for supporting the classification, comparison, verification, evaluation, and improvement of processes. Systematic, rigorous and automatable comparison techniques can help achieve the goals of process engineering.

In their earlier work on this topic, Xiping Song and Leon Osterweil proposed techniques and structures for a disciplined and rigorous software process comparison, and demonstrated their use by carrying out classifications and comparisons of processes drawn from the narrow and specialized domain of software design processes [11], [12]. These comparisons were guided by a formal comparison process, Comparison of Design Methods (CDM), and were performed according to a fixed base framework. The base framework can be thought of as a classification schema and provides guidelines

for grouping comparable activities, artifacts, and features.

The accuracy of such a comparison is limited by the accuracy of the compared process models. It is very likely that any process comparison method is limited by the accuracy of the models. While the CDM was one of the first steps toward systematic process comparison, it was performed manually without any automation. The classification of activities and artifacts relied on the subjective interpretation by the comparer of the names and model descriptions.

Our work builds on experience with the CDM by using the concept of the base framework, a systematic comparison process, and formal process models. It must be noted that structure, content, and construction method of our BF, comparison process, and process models are different from the CDM. The improvements, over the previous work in this area, we strive to achieve are increased objectivity and greater repeatability of comparison results, more focused comparisons, more general applicability than the domain of software development methods, addressing the ambiguity introduced by human involvement in process execution, and partial automation of the comparison process.

The suggested comparison approach assumes that the compared entities (software processes or applications) are in the same problem domain and have a similar purpose, but might have certain differences in how they achieve their goals starting from the input of the same kind and providing output of the same kind. The approach is based on the comparison of artifacts produced by the compared processes along the execution paths prompted by the same input (e.g. the same formal requirements for a software system fed to different software development processes). Thus the approach also makes an assumption that the paths traversed through the compared processes in response to the same input are comparable and produce comparable artifacts.

In this paper we describe a motivating example from the area of telecommunications logistics processes, our comparison process, and the testbed design.

In Section 2 we describe related work. The motivating example is introduced in Section 3. The comparison process is introduced in Section 4. Section 5 describes some details of the implementation for the comparison process testbed. We conclude with description of future research directions in Section 6.

2. RELATED WORK

The work by Xiping Song and Leon Osterweil ([12]) was aimed at comparing software processes rigorously modeled in software process languages according to a base framework (BF) specifically defined for the CDM. The decomposition units of activities and artifacts in their method were classified according to the BF by manual comparison of the description of an activity and/or artifact in the process model or the original source with the description of the BF class. The further comparison of the so classified and hence comparable process decomposition units of the two compared processes was also done manually, guided by a comparison process formalized at a rather high abstraction level. Our approach also uses the concept of a base framework, though its structure, content, and construction method are different from the one suggested by the CDM process. We chose a flat framework that, initially, only includes classes for atomic elements of artifacts (as per a process source description) from a specific problem domain. The comparison approach provides for process decomposition unit groupings at a later stage. Our comparison process is generalized for use with processes from different problem domains rather than being focused on software design methods as the CDM.

The need to compare modeled processes according to a fixed base framework was also recognized somewhat earlier by Sjaak Brinkkemper et al. ([7]). However, their comparison had no guidelines as explicit and formal as the CDM. The BF suggested in ([7]) has a flat structure as well. The content and construction method are different from the BF in our approach. The BF classes in ([7]) are constructed on the basis of the elements of the process and artifact decomposition units of the compared processes. Instead we treat our BF as a pre-defined ontology for a specific problem domain onto which we initially map only those artifact units that need to be examined for a focused comparison. Thus the BF construction in our approach is incremental and it is guided by the comparison topics instead of an across-the board comprehensive comparison.

Both of these approaches used process models based on the original descriptions of the process authors. Thus their comparison is as accurate as the process models are sufficient in accuracy and elaboration. The comparison is also as complete as the base framework and the process models allow them to be.

Analysis of in-place software processes and measurement of the correspondence of a particular process execution to its model have similar goals with process comparison in that they attempt to evaluate processes. Some fairly recent work in these directions has been done by Jonathan Cook and Alexander Wolf ([4], [5], [6]).

3. MOTIVATING EXAMPLE

As our motivating example we used a telecommunications ordering process employed by Telcordia. The ordering process elaborates the activity of adding a service to a customer. This company uses a proprietary process specification language for rigorous specification of such logistics processes. Their logistics processes also use a predefined set of artifact formats. In addition to the format specification, there is a set of well-formedness conditions defined for the artifacts. One of the challenges the developers of these processes face is the task of change management. After a change is made to the process the developers have to make sure that the process still produces artifacts complying with the well-formedness conditions. If the new version of the process produces an undesirable result then the developers have to find out the cause. This is not always as trivial as it would seem even for relatively small processes as not a single developer understands the process in its entirety. There are also different possible interpretations of the process by developers. The suggested comparison approach alleviates some of these problems by providing a rigorous analysis of process artifacts and suggesting possible causes for the differences based on such an analysis.

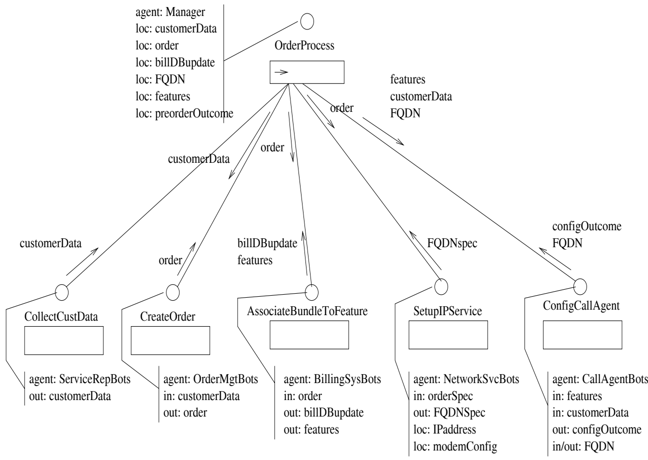

Figure 1: Order process

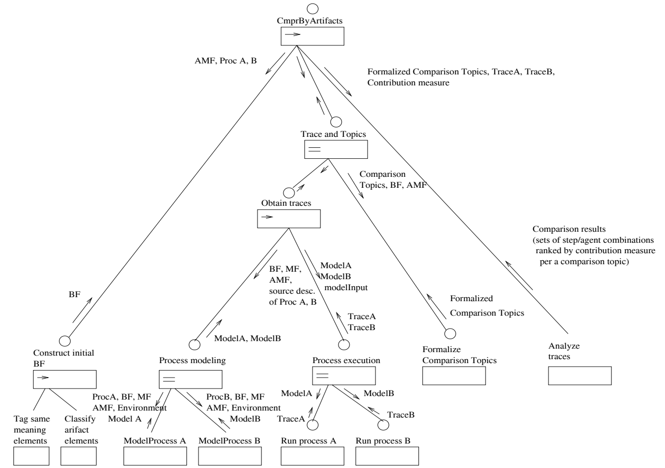

Figure 2: Comparison process

The study assumed that two seemingly identical versions of the same process need to be compared to find out if the artifacts produced comply to a certain well-formedness condition, and to point out the reason for the differences if there are any. Such a set-up is likely to highlight the benefits of the artifact-based trace analysis technique that can be used to complement the static analysis of the process template specification such as by Jamieson Cobleigh et al. ([3]).

The representation of the motivating example process template is depicted in Fig. 1. This figure shows a functional decomposition of process steps. The steps' interfaces include specification of an agent class (*agent:* prefix)[3], local parameters (*loc:* prefix), input parameters (*in:* prefix, and output parameters (*out:* prefix). The data flow is depicted along the decomposition links: the inscriptions near the arrow into a step contain input parameters and that near the arrow out of a step contain output parameters. A complete process specification also includes the resource model that specifies the agents available in the environment, the artifacts specification, and the agents' problem solver components specification that define the transformations from input artifact formats to output ones. The process program declares the agent classes for steps. The actual agents are bound to steps during process execution, therefore it is possible to run the same process template in different environments.

An example of a well-formedness condition for this telecommunications ordering process is the need to base voice communication service on a data communication service. If the ordering process does not establish that a customer ordering the voice communication also needs the data communication then the process creates malformed artifacts that result in billing the customer for the voice service that will not function. To avoid this scenario the executing software process (including the template and functionality of the agents responsible for performance of the steps) has to be shown to comply with the well-formedness condition. Any differences and their possible causes must be found, be they in the process template or agent functionality, and must be reported. Our comparison approach suggests a rigorous and automated way to provide these results.

[3]An agent is an entity responsible for execution of a step.

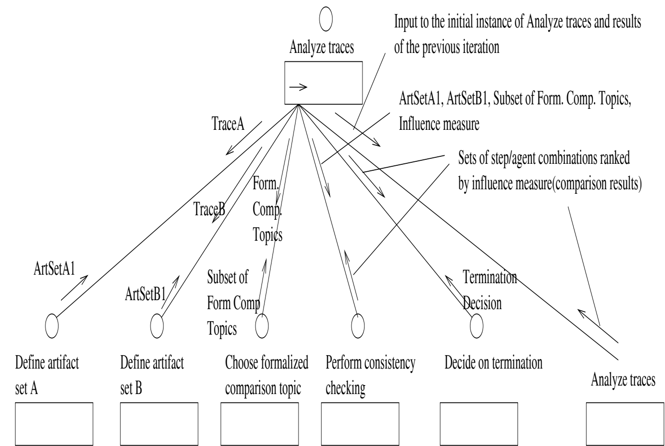

Figure 3: Analyze traces

4. COMPARISON PROCESS

In this section we will describe our comparison process. The description of the comparison process will refer to the diagram of its Little-JIL representation in Figs. 2, 3.

The current implementation of the process is intended for artifact-based comparison of two software processes. The software processes are assumed to have structured artifacts with predefined formats such that the processes specify transformations between the artifact formats.

The comparison process starts with constructing an initial base framework (BF) for the artifact section (step **Construct initial BF**). The step is decomposed into the **Tag same meaning elements** and **Classify artifact elements** substeps to be executed sequentially. This step has a substantial subjective involvement of a human comparer. The BF can be constructed either from an existing ontology or it

is generalized from the artifact formats of the compared processes. The goal of the comparer is to identify the semantically overlapping portions of the artifact formats and tag the semantically similar elements of those formats. This is done in the **Tag same meaning elements** substep based on the source descriptions of the processes **Proc A** and **Proc B**. The output of this substep is a table of correspondence of artifact elements from the original process descriptions and their common naming. The correspondence is needed only between artifact elements in the overlapping portion of the semantics of artifacts. Such an overlapping is likely to exist in processes from the same problem domain and with the same purpose.

In the case of our process implementation we used a common artifact meta-format (**AMF**) and the artifact element naming conventions for tagging. Thus artifact elements are classified according to the AMF (step **Classify artifact elements**) and artifact elements with the same meaning are named the same in the process models and artifacts of the same class. The **Tag same meaning elements** substep precedes the **Classify artifact elements** since it is beneficial to reduce the number of elements to be classified. This reduction is the result of giving the same names to the elements with the same meaning, so the classification decision is made only once for both same named elements from different processes.

Once the artifact section of the BF is defined, the modeling of the processes in the same executable process modeling formalism can proceed. The input to this step includes the base framework (**BF**), process modeling formalism (**MF**), artifact meta-format (**AMF**), and the source description of the compared processes **Proc A** and **Proc B**. It is preferable to feed rigorous specifications of processes elaborated to the level of manipulation of the lowest level decomposition units of artifacts.

This step is further decomposed into modeling of the individual processes that can proceed in parallel. This step requires substantial human involvement and it can be automated in the case if the source descriptions are rigorously defined by building a translator from the formalism used in the source descriptions to the common formalism used for comparison.

The expressiveness of the process formalisms can influence the comparison results if they do not allow modeling of the artifact elements or steps that manipulate them related to the comparison topics. The output of the modeling step consists of the process models in the common modeling formalism (**ModelA, ModelB**). In our implementation we use the Little-JIL as the common modeling formalism for process comparison. Thus the modeling involves representation of the functional decomposition of the process, specification of the process step interfaces, specification of the artifact formats in the AMF, specification and development of the agents to execute the steps, instrumentation of the agents per a step kind, specification and development of the step-specific GUIs, and the definition of the environment to be the same for both processes (the developed agents are included into the environment).

The original Little-JIL has been extended to generalize the agent and instrumentation specification for individual problem domains. The comparer must take care not to overspecify the agents beyond the elaboration of the lowest level activities from the source processes. If the source processes assume certain common low level activities then it is advisable to use the same implementation for the agents from both processes. The extended Little-JIL agent architecture allows for reuse of agents' problem solving components. The Little-JIL artifact specification and the agents must use the artifact formats specified in AMF and complying with the naming conventions for the artifact elements with overlapping semantics.

The comparison process is limited by the level of elaboration of the source processes. If the source process does not describe the activities at the level of manipulation of artifact elements then this method is unlikely to be applicable. The generalized instrumentation components simplify the comparer's task in the process modeling stage. Nevertheless, the comparer must make subjective decisions regarding continuity of the artifact concerns. The comparer must decide on the kind of operation a given agent performs on a given artifact element when performing a certain step ($\boldsymbol{Operations = (Create, Derive, Retain, Modify)}$). Thus every agent, when executing, would add an entry to the annotation lists of the output artifact elements explaining the operation it performed on that element and noting agent and step IDs and the timestamp. Also, the comparer must decide which output artifact elements are going to inherit the annotation lists from the input artifact elements. It is this decision that ensures the continuity of artifact concern traces. It is likely that specifics of a given problem domain might simplify this task. For instance, in logistics processes there is often a limited, predefined set of artifact formats with predefined and explicit relationships between elements from artifacts of different stages of a process.

In software development processes it is also possible to identify the portions of different artifacts that correspond to the same concern. For example, it is possible to trace the artifact elements describing the evolution of the same functionality of the developed system from requirements to design and architecture specifications. Such domain specific knowledge can aid the comparer in making decisions during the instrumentation phase. An example of a process model for the motivating example is given in Fig.1 while Fig. 7 shows an example of an artifact specification using a graph-based AMF.

The next step of the comparison process, **Process execution**, requires execution of the so modeled and instrumented processes (**ModelA, ModelB**) on the same input (**modelInput**). The result of such an execution is a set of two traces of artifacts whose elements are annotated with a list of operations, agents, and steps that were performed on them. The annotation lists in an artifact would cover the trace until this artifact is produced. Thus product artifacts would contain the most comprehensive annotation lists. The annotations of artifact elements are partially ordered by timestamps by construction via the instrumentation code that is run during the process execution. Thus every artifact

```
...
<consistencyrule id="wellform1">
  <header>
    <description>
      Voice service should be associated to data service
    </description>
  </header>
  <forall var="vs" in="$voiceservices">
      <exists var="l" in="$associations">
             <and>
                <equal op1="$vs/@name"
 op2="$l/@source"/>
 <exists var="ds" in="$dataservices">
     <equal op1="$ds/@name"
      op2="$l/@dest"/>
                 </exists>
     </and>
      </exists>
  </forall>
</consistencyrule>
</consistencyruleset>
```

Figure 4: Comparison topic example

element relevant to the comparison topic[4] must have a history of all manipulations done to it in the annotation list. This step outputs the traces of artifacts with annotation lists (**TraceA, TraceB**). The traces follow the execution paths through the process models **ModelA, ModelB** that correspond to the same input **modelInput** and hence are considered comparable.

The step for definition and formalization of comparison topics (**Formalize Comparison Topics** step) can be executed after the initial BF is constructed and in parallel with the **Process modeling** and **Process execution**. This is reflected by auxiliary decomposition steps **Trace and Topics** and **Obtain traces**. This step implies specification of comparison topics in terms of first order logic formulas operating on the artifact elements with common naming conventions. This step outputs **Formalized Comparison Topics** as a set of first order logic formulas. In the case of the motivating example the comparison topic is whether both processes fulfill the requirement that a voice service must rely on an existing data service in the customer's service configuration. This requirement is reflected in a relationship from the voice service to the data service in the *billDPupdate* artifact. One version of the process checks for the data service and establishes the necessary relation. The other version omits this action and produces a malformed artifact which would lead to a failure of the service request set-up in a deployed telecommunications process. This comparison topic is formalized as a first order logic rule in the Xlinkit rule specification language ([2]). The formalized comparison topic is phrased as $\forall \boldsymbol{vs} \in \boldsymbol{voiceservices}\ \exists \boldsymbol{link} \in \boldsymbol{associations}$ s.t. $\boldsymbol{link.source} = \boldsymbol{vs} \vee \boldsymbol{link.destination} = \boldsymbol{ds}, \boldsymbol{ds} \in \boldsymbol{dataservices}$. The Xlinkit rule specification of the comparison topic is shown in Fig. 4.

Next, the comparison process calls for analysis of artifact traces **TraceA, TraceB** by way of consistency checking to the formalized comparison topics. This analysis is done in the **Analyze traces** step. The step's input consists of **Formalized Comparison Topics**, annotated artifact traces **TraceA, TraceB**, and **contribution measure**. The contribution measure quantifies the degree to which a certain step of a compared process influences the result of consistency checking to a particular formalized comparison topic. For instance, the difference between the number of consistency links from a formalized topic to the step's output artifacts and the number of consistency links to the step's input artifacts might serve as a measure of the step's contribution. The step's output forms the results of the artifact-based comparison - consistency links between formalized comparison topics and process artifact elements and sets of step/agent combinations ranked by the contribution measure per a comparison topic. One of the main outcomes of such an analysis is comparison of consistency links from the same comparison topic to artifact elements in different processes. The consistency links help highlight whether:

- both comparison processes comply with a certain comparison topic,
- both comparison processes violate a certain comparison topic,
- one process complies with a certain comparison topic while the other violates it.

The annotation lists help point out the steps and agents that are responsible for the analysis outcome. In addition, it is very likely that the sets of step/agent combinations from different traces (processes) corresponding to the same comparison topic are comparable. Such pairs of sets can give insights about the functional similarities or differences between processes and aid the comparer in making a more objective process comparison.

The functional decomposition of the **Analyze traces** step is elaborated in Fig. 3. The analysis essentially consists of iterative consistency checking of sets of artifacts against the formalized comparison topics. The next artifact set is determined by the annotations of the artifact elements from the previous artifact set. The analysis subactivities are described below.

4.1 Choice of initial artifact sets

The set of product artifacts quite often is most useful to be used as the initial artifact set because they contain the most comprehensive annotation lists and they are likely to contain all the artifact elements that need to be checked for comparison topics. Another advantage to this choice of the initial set is high likelihood that the product artifacts are explicitly defined in the process source description. To reduce the amount of computation it is advisable to choose artifacts that are known to be relevant for the chosen comparison topics. Thus, the initial artifact sets **ArtSetA1** and **ArtSetB1** are defined by the steps **Define artifact set A** and **Define artifact set B**.

4.2 Choice of comparison topics

Once the artifact sets are chosen, the comparer should choose the comparison topic to be used for consistency checking. This is done in the **Choose formalized comparison topic** step.

[4]the one that needs to be checked in order to determine if an artifact complies with a certain comparisons topic

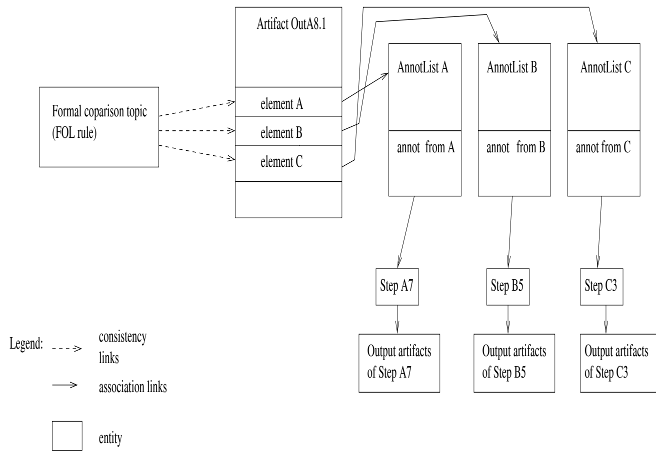

Figure 5: Step and artifact relationships for the next artifact set formation

4.3 Checking of artifacts' consistency to formalized comparison topics

Once the initial artifact sets for both processes and the formalized comparison topic are chosen, the comparison process runs a consistency checker that produces consistency links between the formalized comparison topics and artifact elements of the compared processes. The current implementation of the comparison process uses the Xlinkit consistency checker by Christian Nentwich et al. ([2], [9]). Next, the comparer analyzes the two sets of artifact elements that have consistency links to the same formalized comparison topics, but belong to different processes.

4.4 Choice of next set of artifacts for consistency rules

If the comparer wants to assess the contributions of all the steps responsible for the analysis outcome then the consistency checking must be done on all relevant artifacts along the step/agent path. This path is stored in the annotation lists of artifact elements pointed to by consistency links from one comparison topic. Thus an iterative analysis has to be performed and the next set of artifacts has to be chosen. The iteration termination decision is made in the **Decide on termination** step. The analysis might stop once the annotation sets for all related artifact elements are exhausted or it might terminate earlier, once the artifacts with last modifications/additions to the related artifact elements are reached.

The next set of artifacts to be compared is determined from the annotations lists in the following way. For a given set of artifact elements, the process retrieves the annotations that correspond to the immediately preceding manipulation and retrieves from the annotations the steps responsible for the manipulations. For example, let us suppose that the latest consistency checking produced links from a certain formal comparison topic to the elements **A, B, C** of artifact **OutA8.1** (as in Fig. 5). Each of these elements also has an associated annotation list. The penultimate elements of these lists will contain annotations describing the immediately preceding manipulations on these elements. From these annotations we can obtain the step IDs that identify individual runs of process steps (e.g. A7, B5, C3 in Fig. 5). It's the output artifacts of these steps that can be used to form the next set of artifacts. The steps A7, B5, C3 are not necessarily immediately preceding the step A8 that produced the OutA8.1 artifact because some artifact elements might have been passed on by some steps without modification.

5. OVERVIEW OF THE DESIGN EXTENSION TO THE JULIETTE ENVIRONMENT

In this section we will describe one possible way of implementing the comparison process testbed. The ability to execute a rigorously modeled software process is required by the process. The comparison process also assumes that a software process specification is viewed as a template that is instantiated and run in a specific environment that includes agents, among other resources needed by processes. Agents are assumed to be capable of performing different portions of the process. Thus, one of the most important components of our implementation is a software process execution environment that uses the notion of agents.

We chose the Juliette process environment and the associated Little-JIL process specification language ([1], [13]). The Little-JIL process language has been designed from the start for specification of software processes and it incorporates a number of features that take into account the peculiarities of software processes. To name a few, these features include proactive and reactive control specification, the use of a multi-agent approach to model software process environment, specification of communication, coordination and collaboration between the agents, orthogonal specification of resource environment. The Little-JIL process specifications can be thought of as process templates. The Julliete process environment has been designed to instantiate Little-JIL process specifications.

This choice is due to a sufficient expressiveness of the Little-JIL language for our purpose and the use of the notion of an agent in the Little-JIL and Juliette. Nevertheless, the implementation of the comparison approach required two main extensions to be made to the Juliette environment: a problem-domain specific agent framework and a generalized artifact specification. These extensions allow for unified specification and reuse of the whole agents and their problem solving components[5], unified code instrumentation, and common artifact specification (via an Artifact Meta-Format) required for artifact-based comparison.

The addition of the agent framework allows for greater software reuse, greater focus of agents' definitions and hence their greater comprehensibility, and it also reduced the time it takes to define new agents in the same problem domain. Essentially, by restricting the structure of an agent and providing agent templates we gain a great deal of leverage in

[5]This feature of the agent framework is important for consistency of comparison because the compared processes might specify the same activity to be performed. This would most likely be the case if the comparison is done between different versions of the same process for the purpose of change management

defining agent-based processes.

The use of Artifact Meta-Format (AMF) enables us to define artifacts according to the chosen common artifact section of BF which is the cornerstone of the comparison analysis. In addition, the use of AMF allows for unified handling of artifacts by agents and facilitates generalized instrumentation.

5.1 Agent framework overview

The Juliette software process environment requires only that agents be able to listen to agenda item events. No guidelines are given as to the structure of the agent and its problem solving component. Neither are there any guidelines about the representation of artifacts. The software process designer using the Little-JIL is left to make his/her own decisions regarding the structure of the agents and regarding the format of the communication between them. After specifying several software processes and building agent structures from scratch a software process designer is likely to recognize that it might be beneficial to introduce more constrained frameworks that provide a predefined structure for the agents that spells out how to go about constructing an agent and enables the reuse of the agents originally written for other processes in the same problem domain. It is understandable that the freedom the stock Little-JIL provides is needed because it is impossible to envision and generalize all areas of application. On the other hand, once such areas of application are recognized, it is beneficial to introduce extensions to the software process environment specific to that application area.

This reasoning led to development of the agent framework for problem domain specific processes shown in Fig. 6. The framework has been developed in Java with the use of XML for artifact meta-format specifications.

5.1.1 Agent framework structure

The framework consists of a hierarchy of classes for the definition of agent components at different levels of generalization: the AutoAgent class, process specific agent, StepProduction, and GraphStepProduction classes. The framework also includes the StepGUI class for step specific GUI specification, ArchGraph class as a container for a generalized artifact definition via a graph-based Artifact Meta-Format and artifact reader and writer classes.

The AutoAgent and process specific agent classes are to be inherited by the user agents. The StepProduction and GraphStepProduction classes are to be inherited by the problem solving components that are parts of a user agent. A problem solving component defines a certain low level activity that an agent can perform. In case of software processes that transform artifacts from one format to another a problem solving component corresponds to specification of an artifact format transformation.

The AutoAgent class generalizes functionality for matching the step that the agent is requested to perform with an activity that accomplishes the step. This class implements mapping of step names to problem solving components, artifact storing, and interface with the Little-JIL language interpreter.

Some methods implemented by the AutoAgent class:

- *protected void initialize(String[] args, AutoAgent agent)* - initialization of the agent environment: agent registration, step mapping initialization, self-localization in the file system, determination of the paths for artifact storage
- *public void storeStepParam(AgendaItem item, ...)* - generalized storage of artifacts for forming an artifact trace per an agent
- *posted (AgendaEvent evt)* - making a decision whether this agent is able and willing to execute the step corresponding to the given agenda event

The process specific agent class obligates its subclasses to have a certain common functionality set and a certain common structure specifically suited for process agents from a given problem domain. Part of that common functionality provides for communication with the external environment, in this case - with other components of the Juliette software process environment (such as the Little-JIL language interpreter). The framework also provides for problem solver components interchangeable between the agents. The set of problem solving components defines the functionality of a particular agent. The process specific agent invokes problem solving components in response to the step related events from the Little-JIL language interpreter.

Depending on the degree of the desired automation the agents can be:

- Human assistants (such an agent would invoke a step-specific GUI and try to assist the human in accomplishing the step)
- Human-modeling (such an agent would attempt to model the duties of a human for process simulation or guidance purposes)
- Automated (such an agent would accomplish steps amenable to complete automation)

The abstract StepProduction class declares a general functionality for a problem solving component which is an *Object execute(Object[] args)* method. Thus the framework forces many to one artifact transformations for the atomic activities of an agent. The GraphStepProduction class enforces the use of the graph-based format for the artifacts that problem solving components manipulate. The domain and/or process specific problem solving components are to extend the GraphStepProduction class with implementation of specific activities or artifact transformations. An agent instance instantiates the set of process specific problem solving components that define that agent's functionality and calls the *execute* methods on them in response to the agenda events from the process language interpreter.

Any software process is likely to require step specific GUIs because of human involvement in the process execution. This need is recognized in the agent framework by defining a set of step specific GUIs that is checked by the agent every time it receives a new agenda item.

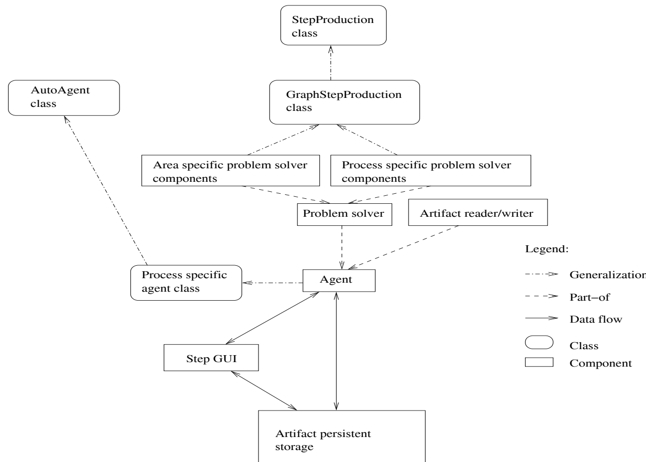

Figure 6: Agent structure

5.2 Artifact Meta-Format overview

The Artifact Meta-Format introduces a graph-based generalization for the artifact formats. It allows definition of a class hierarchy of nodes, links, and their attributes. The meta-format supports inheritance of node and link attributes by type and value. The meta-classes of the meta-format are:

- **MetaComponentClass** - a node class
- **MetaComponentInstance** - a node instance
- **MetaLinkClass** - a link class
- **MetaLinkInstance** - a link instance

The entries of the attribute lists of the meta-class instances are defined by the tuples of their class (attrClass), name, and value.

The artifacts defined in the graph-based AMF are stored persistently as XML files. The corresponding Java framework allows instantiation of a persistent artifact representation via instances of Java classes. The AMF reader class can be thought of as a graph-based AMF class-loader. The framework also provides a writer class that provides functionality for storing artifact instantiations. A visual editor has been developed to support the execution of processes that use this AMF for artifact specification.

An example of a persistent artifact representation is given in Fig. 7. This figure depicts a portion of the *customerData* artifact which was produced by the agent *ServiceRepAgent* when executing the *CollectCustData* step on run 1045706480. The tags *MetaComponentClass, MetaComponentInstance, MetaLinkClass, MetaLinkInstance* correspond to meta-classes of the AMF. The artifact graph classes and instances are attributed. The attributes are specified with triples of attribute class, name, and value. The AMF supports inheritance of attributes by types and values.

6. CONCLUSIONS AND FUTURE WORK

This line of research was undertaken in reaction to a long string of process comparison work that was completely informal, offering no basis for scientific validation through reproducible experimentation. It was our goal that process comparison be made rigorous, semantically well-founded, and reproducible through the use of formally defined comparison processes (such as the CDM), comparison schemas (such as BF), and semantically well-based modeling formalisms. This work continues to provide evidence that this sort of rigor and reproducibility is possible.

There are also other avenues of further work worth pursuing, particularly that of experimenting with an approach in non-software process domains. One of such experimentation directions is the application of the comparison process to the evaluation of path specific optimization techniques for user-guided applications (such as GUIs). The authors of continuous optimization techniques ([8]) assume that a user is likely to make the application traverse the same execution path when the same user accomplishes the same kind of task. Hence the optimizer takes the traversed path into account when scheduling the instructions. Our approach can help in data-based evaluation of the optimizers by keeping track of instructions that were applied to elements of application's data structures. Thus it would be possible to show different emphasis the optimized execution paths put on application's data elements and explain possible variance in performance of the optimizers. These comparison results might be used by AI techniques for choosing the appropriate path-specific optimization technique during an application's execution. Another experimentation direction is application of the comparison to more extensive processes.

The comparison process itself can be evolved. One such direction is a more generalized set of possible kinds of artifact elements transformations based on the nature of artifacts. Once such a set is identified for a certain problem domain, the agents for process steps of this domain could be assembled from such a basic set of manipulations. Identification of such a set would also help an automated elaboration of the compared processes to the same level of specificity which will further aid the comparison by parceling out the comparable and functionally analogous portions of the steps from the compared processes.

7. REFERENCES

[1] Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, Stanley M. Sutton, Jr., and Alexander Wise. Little-JIL/Juliette: A Process Definition Language and Interpreter. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland*, pages 754–757, June 2000.

[2] C.Nentwich, L.Capra, W.Emmerich, and A.Finkelstein. xlinkit: a consistency checking and smart link generation service. In *ACM Transactions on Internet Technology, 2(2)*, pages 151–185, May 2002.

[3] Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil. Verifying Properties of Process Definitions. In *Proceedings of the ACM Sigsoft 2000 International*

```
<?xml version="1.0" encoding="UTF-8"?>
<T1003050159763.Agent.ServiceRepAgent.Step.CollectCustData.OutParam.customerData.run.1045706480>
    <Node>
        <MetaComponentClass>
            <Attribute attrClass="java.lang.String" name="name" value="customerData"/>
            <Attribute attrClass="java.util.Hashtable" name="children" value="Customer1223027"/>
            <Attribute attrClass="java.lang.String"
                name="customerPhoneNumber" value="000-000-00-00"/>
            <Attribute attrClass="java.lang.String" name="defaultName" value="defaultValue"/>
            <Attribute attrClass="java.lang.String"
                name="customerStreetAddress" value=""/>
            <Attribute attrClass="java.lang.String"
                name="customerZipCode" value="11111"/>
        </MetaComponentClass>
    </Node>
    ...
    <Node>
        <MetaComponentInstance>
            <Attribute attrClass="graph.model.ComponentClass"
                name="class" value="customerData"/>
            <Attribute attrClass="java.lang.String" name="name" value="Customer1223027"/>
            <Attribute attrClass="java.lang.String"
                name="customerPhoneNumber" value="617-234-92-32"/>
            <Attribute attrClass="java.lang.String" name="customerName" value="Edward Jackson"/>
            <Attribute attrClass="java.lang.String"
                name="customerStreetAddress" value="962 Hill Dr."/>
            <Attribute attrClass="java.lang.String"
                name="customerZipCode" value="01403"/>
        </MetaComponentInstance>
    </Node>
    ...
    <Node>
        <MetaLinkClass>
            <Attribute attrClass="java.lang.String" name="name" value="association"/>
            <Attribute attrClass="java.util.Hashtable" name="children" value="Customer1223027RequestsServiceReq8745"/>
            <Attribute attrClass="java.lang.String" name="type" value="association"/>
        </MetaLinkClass>
    </Node>
    <Node>
        <MetaLinkInstance>

            <Attribute attrClass="java.lang.String" name="class" value="association"/>
            <Attribute attrClass="graph.model.ComponentInstance"
                name="source" value="Customer1223027"/>
            <Attribute attrClass="graph.model.ComponentInstance"
                name="dest" value="ServiceReq8745"/>
        </MetaLinkInstance>
    </Node>
</T1003050159763.Agent.ServiceRepAgent.Step.CollectCustData.OutParam.customerData.run.1045706480>
```

Figure 7: customerData artifact in graph-based AMF

Symposium on Software Testing and Analysis (ISSTA 2000), pages 96–101. Portland, OR, August 2000.

[4] Jonathan E. Cook, Lawrence G. Votta, and Alexander L. Wolf. Cost-Effective Analysis of In-Place Software Processes. *IEEE Transactions on Software Engineering*, SE-24(8):650–663, August 1998.

[5] Jonathan E. Cook and Alexander L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[6] Jonathan E. Cook and Alexander L. Wolf. Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model. *ACM Transactions on Software Engineering and Methodology*, 8(2):147–176, April 1999.

[7] Sjaak Brinkkemper Geert van den Goor, Shuguang Hong. A Comparison of Six Object-Oriented Analysis and Design Methods. Technical report, University of Twente, Enschede, the Netherlands, 1992.

[8] Thomas Kistler and Michael Franz. Continuous Program Optimization: Design and Evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[9] C. Nentwich, W. Emmerich, and A. Finkelstein. Static Consistency Checking for Distributed Specifications. In *Proceedings of Automated Software Engineering 2001, San Diego, USA*, 2001.

[10] Leon J. Osterweil. Software Processes are Software Too. In *Proceedings of the Ninth International Conference of Software Engineering*, pages 2–13, Monterey CA, March 1987.

[11] Xiping Song and Leon J. Osterweil. Engineering Software Design Processes to Guide Process Execution,. Technical Report TR–94–23, University of Massachusetts, Computer Science Department, Amherst, MA, February 1994. Appendix accepted and published in Preprints of the Eighth International Software Proces Workshop.

[12] Xiping Song and Leon J. Osterweil. Experience with an approach to comparing software design methodologies. *IEEE Transactions on Software Engineering*, 20(5):364–384, May 1994.

[13] A. Wise. Little-JIL 1.0 Language Report. Technical report 98-24, Department of Computer Science, University of Massachusetts at Amherst, 1998.