Software architecture for flexible integration of process model synthesis methods

Rodion Podorozhny
Texas State University
rp31@txstate.edu

Anne Ngu
Texas State University
hn12@txstate.edu

Dimitrios Georgakopoulos
Telcordia Research
dimitris@research.telcordia.com

Dewayne Perry
The University of Texas
perry@ece.utexas.edu

Abstract

In this paper we suggest an architecture that would integrate various methods for synthesis of a software process model based on domain knowledge about artifacts, process fragments, tools, and limited process execution observations. Our approach suggests using a meta-process specification for integration of various process synthesis methods to provide a generalized process model. We also propose using a process execution observation for confirmation of a synthesized process model.

1. Introduction

The area of software engineering is mainly concerned with methods for synthesis and analysis of software. In this paper we are focusing on an automated method for synthesis of software processes based on very few observations and various domain knowledge. The automation of synthesis brings many benefits: repeatability, efficiency, cost reduction. In addition, a process synthesized by our approach is itself automated due to the use of learning methods for synthesis of control flow decisions. Some of the early work on process discovery is based on analysis of retrospective data about the process executions [1,2,3,4]. The process models discovered with these early methods focus on the set of activities and the process control flow. The process synthesis method integration we suggest expands the earlier process discovery work in several directions. Our approach is novel because it allows to automatically synthesize a process based on very few observations of process execution (process instance) thanks to the use of learning methods and a greater variety of domain knowledge. It provides a process model enhanced with activity attributes and control flow decision-making.

Our approach has been informally evaluated in the context of a real problem of a human-centric process of technical object control and was considered technically feasible, useful, and practical.

In this work we are focusing on the *architecture for integration* of methods for synthesis of various process aspects that comprise a process model as opposed to the details of the methods themselves. The architecture assumes availability of the domain knowledge about the process artifacts, tools to be used by a process, the environment in which a process is to be enacted, and a limited number of observations of process execution. In a way, we propose an approach that involves synthesis of a process model via limited observations of examples of its execution.

2. Problem statement and approach intuition

The problem calls for a synthesis of a *rich* process model based on a *single observation* of process execution. The real world need to solve this problem arose due to the need to learn plans or processes from human users by being shown one example. Sample applications include air-tasking-order planning and CAD design planning. Such a process learner will learn from one example by opportunistically assembling knowledge from many different sources, including generating it by reasoning. The process model must be rich because the knowledge of a greater number of process aspects increases the accuracy of analysis, lends better process improvements, allows for the use of a greater variety of domain knowledge about the learned process. The assumptions include availability of domain knowledge including artifact well-formedness constraints, bill of materials, generalized description of a finished product or service, tools used for process execution, process fragments, environment conditions for process execution and others. Each of these kinds of information can be either *incomplete* or have some degree of *uncertainty.*

Next let us give an explanation of our intuition about a possible solution. Abstractly, this problem seems to require us to synthesize an accepting computation abstraction specification (e.g. Turing machine) for a language based on one sentence of that language. The retrospective data about previous executions of this process is assumed to be very limited. How could we produce a generalized process

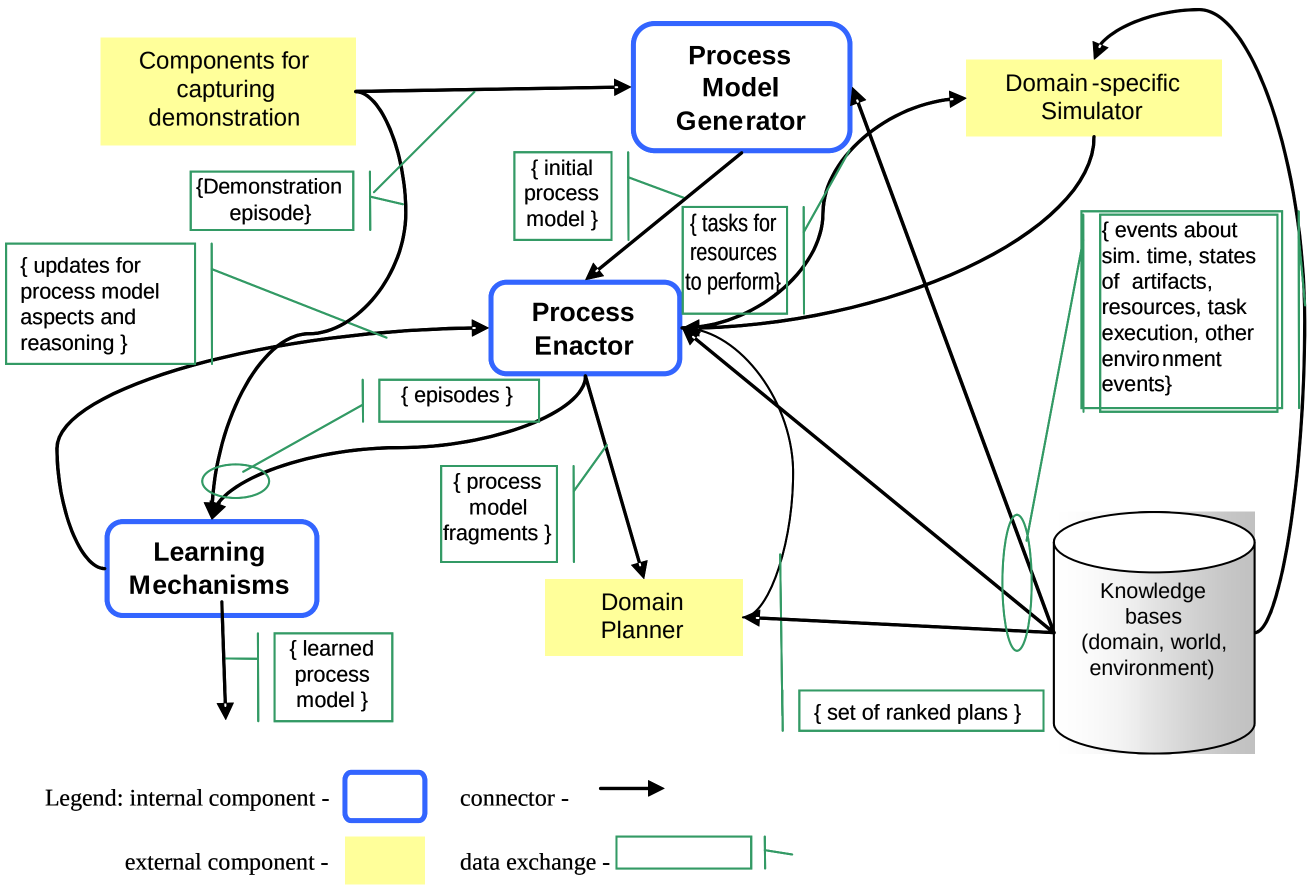
Components for capturing demonstration
Process Model Generator
Domain-specific Simulator
{Demonstration episode}
{ initial process model }
{ tasks for resources to perform}
{ updates for process model aspects and reasoning }
Process Enactor
{ events about sim. time, states of artifacts, resources, task execution, other environment events}
{ episodes }
{ process model fragments }
Learning Mechanisms
Domain Planner
Knowledge bases (domain, world, environment)
{ learned process model }
{ set of ranked plans }
Legend: internal component -
connector -
external component -
data exchange -

Figure 1 Architecture of a process synthesis integrator

specification if we are given only one path which even might be providing an unreliable product ? It seems that doing so without the knowledge about artifacts is impossible due to the lack of information. Thus, our approach prescribes construction of a generalized process model based on whatever domain knowledge is available and then checking whether the observed execution path is a feasible path through our model. It is quite possible that the observed path itself provides a low quality product. Nevertheless, if that path is feasible in the "guessed" process model then it validates the model. Otherwise, the path is considered to be a counterexample, the process model is considered incorrect and thus it is modified to satisfy the observed path.

At the implementation level, we suggest specifying the synthesis process itself in a process language, this allows for flexibility of the integration of methods for synthesis of particular process aspects based on particular domain knowledge available.

3. Process model aspects

In this section we will discuss in greater detail the kinds of information needed to specify a *rich* process model. The final integrated process synthesis system must be able to synthesize the specification of these aspects from available domain knowledge and the observation.

By a process we understand a systematic, disciplined way of either producing a final artifact or delivering a service. Since it is possible to model a service as an electronic artifact we will use the term "final artifact" or product to denote a process outcome.

By a generalized process model we understand a process "program" that, if specified in a rigorous process specification language, can be instantiated by a process enactment engine provided an input and environment specifications. The following aspects characterize a *rich* generalized process model:

- set of activities
- each activity characterized by an identifier, interface (sets of input and output artifact types), pre-condition and post-condition of activities

- constraints on control flow of activities, relaxed to the extent possible
- hierarchical decomposition of activities according to various sets of criteria
- specification of artifact type system manipulated by activities
- specification of resource needs of activities including people roles and tools
- predictors that provide distributions for cost, duration of individual activities and assessment of quality of output artifacts of an activity.

The execution of a process is greatly influenced by guidelines or reasoning mechanisms for control flow choices which are not part of process specification per se, rather they are part of the resources specification. These guidelines define reasoning of resources assigned to execution of process activities. It is our intent that the process synthesis system will discover such guidelines to accompany the generalized process model. The generalized process model must not over-constrain the control flow. That is why a straightforward mapping of the observation onto the generalized process model is unacceptable. For instance, it might turn out that a certain sequence of artifact transformations was enforced by scarce resource availability while in the generalized model those artifact transformations can happen in parallel given abundant resources.

4. Architecture of process synthesis integrator

The previous section outlined the various aspects of a process model that must be ultimately synthesized. Intuitively it is clear that various synthesis methods would be needed to synthesize those aspects. The choice of a particular method depends on the aspect, kind, amount, and certainty of domain knowledge available, nature of domain knowledge, whether the chosen methods are an effective match. Thus the architecture must allow for great flexibility in the choice of the set of activities involved in the synthesis, the tools used, the sequence of their application. Therefore we suggest using a process specification and an associated process execution system for the synthesis process itself.

According to Perry&Wolf [12] an architectural description is comprised of elements, form, and rationale. The architecture of the process synthesis integrator depicted in Fig. 1 contains the elements (sets of components and connectors) and the form (constraints on their interconnections). The architectural description in Fig. 1 uses rounded corner rectangles to denote the internal components, yellow rectangles denote external components, directed arcs to denote connectors, rectangles associated with the arcs to denote the data, the cylinder to denote persistent storage. For the purpose of this paper we limited the architectural description to high level of abstraction without the use of dedicated rigorous architecture description languages such as ACME, Wright or xArch. As a matter of fact this high level architecture also gives an idea about a process specification for the integrator.

The separation of components set into internal to the integrator and external ones signifies the division between the components supplied by domain experts from other organizations and components that are native to the integrator.

Below we will describe the architecture rationale referring to the architecture's elements (components and connectors) from Figure 1.

The Process Model Generator forms an initial process model in a chosen process specification language based on the domain knowledge (product, process, known execution traces, resource utilization) and refines some aspects of the initial process model based on the demonstration episode (observation).

Next Process Enactor, Domain-specific Simulator, Learning Mechanisms, and Domain Planner synergistically subject the initial process model to dynamic analysis and refinement. The interaction of these architecture elements is as follows. The Process Enactor receives an initial process model by data exchange along a connector from the Process Model Generator and, provided the cost, duration, quality of activities can be estimated, submits a fragment of the process to the Domain Planner by the corresponding connector. The Domain Planner chooses the set of process activities from the model and provides partial order for them. It is possible for the Domain Planner to choose an approximated optimal set of activities if their cost, duration, quality are available. If the estimates of cost, duration, quality are not available then the Process Enactor chooses the first set of alternative activities based on the control flow constraints specified in the initial process model. Thus the choice of activities will produce a feasible path through the process model barring the optimality approximation. A planner based on the design to criteria paradigm can serve as an example of a planner that reasons based on cost, duration, quality of activities [16]. For some processes a human can play the part of the Domain Planner as an override of an automatic planner.

If the resource declarations are not available, the activities from this set are either chosen to be performed by the resources themselves (self-identification) or they are assigned based on the matching of the functionality of resources to the nature of transformation of artifacts by the activities. The self-identification implies that resources would request an assignment to activities themselves once they find out from a registry that another activity has been posted for execution. The constraints on the availability of identified suitable resources can further refine the first set of activities to be started.

Next, the Learning Mechanisms might use their exploration method in face of the uncertainty about activities cost, duration, quality to refine the set of alternative activities. The refined set of alternative activities is assigned by the Process Enactor to resources in case the resource declarations have been already identified from the domain knowledge.

Once the set of the activities is identified and resources are assigned, the Simulator starts modeling the execution of the tasks by the modeled resources which affect the state of modeled artifacts. The Simulator notifies the Process Enactor of various events such as completion of tasks by resources, time ticks, contingencies due to the modeled environment, contingencies due to artifact states. The Process Enactor reacts to the events from the Simulator or events generated by the Process enactor itself (e.g. time-out of activity completion). The reactions themselves are specified as process fragments, they can be either domain-specific or default reaction processes (e.g. reaction to time-out of activity completions, resource unavailability contingencies, resolution of contradictions between needs of concurrently running process fragments, pre-emption control decisions).

The Simulator models the transformations of artifacts as these artifacts are processed by resources according to manipulations prescribed by process activities. The state of modeled artifacts is used by the Learning Mechanisms to update the knowledge they accumulated about various aspects of the process such as the control flow decisions, decisions on the sets of activities to be executed and resource assignments. In addition, the Simulator produces estimates of cost and duration based on the models of resources and artifacts, these estimates are also used by the Learning Mechanisms to update the predictors for cost and duration or to suggest new ones in case the domain knowledge contained no such estimates to begin with.

Thus the simulation and learning of a single process instance (called *episode* in the machine learning terminology [10]) continues until either all the activities are finished and/or final artifacts are produced or predefined time runs out or it is determined that there are insufficient resources to produce final artifacts. We will use the terms *process instance* and *episode* interchangeably from now on, meaning a trace of a particular execution of a process model. On completion of an episode the Learning Mechanisms update their knowledge based on the results of an episode.

The demonstration episode results are used by the Learning Mechanisms to update their knowledge. The demonstration information about the cost, duration, quality control flow decisions, resources used are given higher preference over such data obtained from simulation. The number of episodes can depend on the coverage criteria, number of typical environment states in which a process is supposed to function, the amount of knowledge about process executions.

The demonstration episode is used as a test-case to verify the process modeled learned from the domain knowledge and simulation. The process synthesis integrator will attempt to recreate a demonstration episode by simulation based only on the process model, resource knowledge and knowledge of environment changes. We assume that a demonstration might not necessarily result in a well-formed final artifact due to contingencies. So the simulator should be able to recreate the contingencies to check the correspondence of the process model behavior to a real world execution. The flexibility of the choice of a learning mechanism (that would depend on the process aspects to be learned) is allowed by the use of a process execution system. If a different mechanism is required, the process specification for the integrator is modified to include a different step with a different tool invoked.

Considering the space restrictions we cannot provide detailed description of the components of this architecture. Some additional information about the Process Model Generator component and likely candidates for the learning methods is mentioned below.

4.1 Process Model Generator

As our process specification and execution system we chose the industry-level Atlas system developed at Telcordia Research Center in Austin. The Atlas system has already been successfully used in a number of projects [13][14][15]. Its design is tightly connected to a database used for persistent storage of processes and

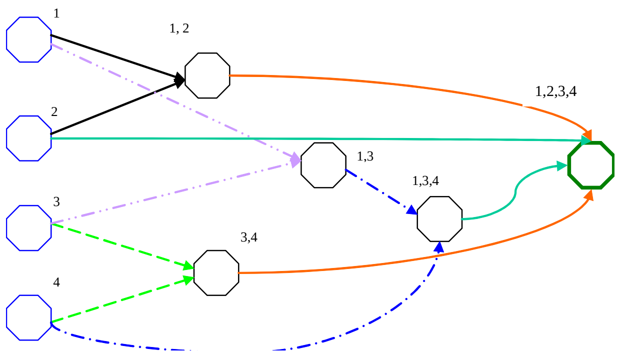

Figure 2. Partial order state space.

process fragments which results in the linear dependency of the scalability of the process execution system on the scalability of the database.

We have some initial experience with the process model generator that uses artifact domain knowledge (i.e. possible artifact transformations) to synthesize such process aspects as the set of activities, their input/output and their execution constraints. The process model generator has been specified in the Atlas specification language enhanced with JPython script that invokes a process generator subcomponent written in Java. This subcomponent uses the specification of the bill of materials and the finished product to guess an initial process model. The bill of materials and the finished products are expressed in an agreed upon artifact ontology. The artifact representation language resembles a software architecture description language in its constructs for specification of artifact components, connectors, ports, and "glue" [12]. The Java subcomponent writes the initial process model in an Atlas process specification.

Ultimately the process synthesis integrator will use multiple generators to construct the set of all distinguishable paths from which we can construct an initial process model. Some of the kinds of domain knowledge we assume would include:

1. Artifact structure and well-formedness constraints on possible combinations of artifact decomposition units. These can determine a subset of the powerset of artifact decomposition unit combinations corresponding to legal combinations, some of which are final artifacts.
2. Resources such as physical tools or automated software tools with known functionality. These can provide legal partial order of tool applications to input and intermediate artifacts that can lead to final artifacts of the process being discovered.
3. Known fragments of the process being discovered. This specification already provides us with an initial, but possibly incomplete, set of distinguishable paths.
4. Traces of execution of the process being discovered. These traces can be generalized into a process fragment [2, 3, 4, 5].
5. Domain-specific intents. These can determine the partial order of sub tasks that can lead to the final artifact

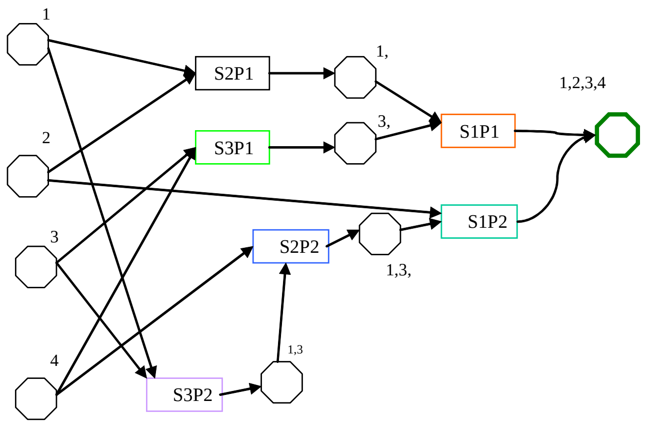

Figure 3. Partial order activity state.

Next we give a high-level description of the principle of operation of the process model generator component of the integrator architecture. This description focuses on the case when the process domain knowledge contains well-defined constraints on the artifact well-formedness. This is especially likely in the domain of mechanical assembly processes. The input to the process model generator is assumed to be a bill of materials, constraints on artifact well-formed interconnections and the description of the desired final product. Essentially, a naive approach of the process generation implies trying all possible ways of putting the available parts together such that they lead to a legal final combination (product of assembly).

Assuming the artifact well-formedness contraints are available, first we generate a graph that describes transitions between all legal combinations of artifact elements. A transition corresponds to a single modification in the set of combined atomic artifact elements. Such a graph is shown in Fig. 2. We call this graph a partial order state space.

In Fig. 2 each state corresponds to subsets of process artifacts that comply with artifact well-

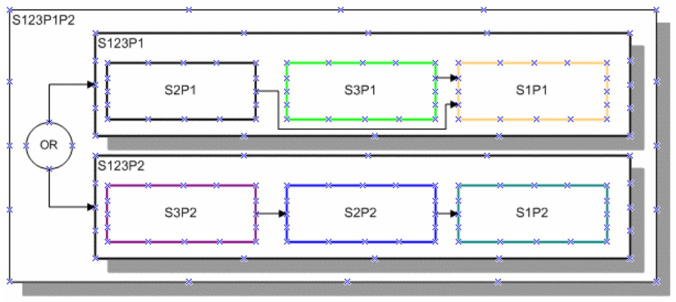

Figure 4. Initial process model.

formedness constraints. It is possible to generate these states based on the bill of materials and the well-formedness constraints of artifact combinations. The sets of transitions between such states would correspond to possible activity instances. The partial order space will also indicate input/output artifact sets for the activity instances. In Fig. 2 we assume there are four product parts that comprise the final artifact(s). Octagons represent states that are marked with the sets of parts. For instance, the blue octagons denote the 4 individual atomic artifacts from the bill of materials and are marked with artifact “1”, artifact “2” and so on. The well-formedness constraints allow for 5 physically possible combinations of these artifacts, one of which is the final product (all artifacts combined). Fig. 2 shows two possible ways of deriving the final product.

Based on the partial order state space, the process model generator must construct a hypothetical general process model. Such a model will explicitly represent the activities that correspond to transitions in the partial order state graph. Thus a partial order activity graph will be produced. It will represent all possible ways to assemble a final product as trees, the leaves of which are the individual atomic artifact elements, the root is the final product and intermediate nodes correspond to legal but incomplete artifact combinations. They are incomplete according to the final product specifcation.

Generation of the set of all distinguishable trees. An initial process model derived from a partial activity order graph should include a complete set of alternative trees, but only a subset of them could be distinguishable due to incomplete domain knowledge. The domain knowledge provides us with constraints of varying degree on product decompositions, partial order of tool applications, process fragments or intents structure (goal tree). There seems to be a relationship between the nature of a process and/or its artifacts and/or its resources and the degree of constraintedness of kinds of domain knowledge. The degree to which any of the different kinds of domain knowledge are constrained determines our choice of which domain knowledge will be used for the initial set of alternative paths.

Ultimately, the partial order of activities in an initial process model is defined by the constraints of the different kinds of domain knowledge. To suggest the alternative trees through a graph corresponding to each partial order of activities, the generator needs to impose these constraints on all possible paths. The first step to generate the partial order efficiently is to apply the sets of constraints (e.g. artifact imposed constraints, tool functionality imposed constraints) to all possible paths in order of decreasing degree of constraintedness. For instance, if the artifact element combinations are more constrained than the resources used to combine the artifact elements then it is the constraints on the artifacts that must be taken into account first. In a scientific process domain, usually top-down proactive process models or fragments are available; therefore they can be used as a starting point for constraining the partial order of the activities space. The set of resources and the sets of their functionality on the other hand are constrained very loosely in this domain. On the other hand, processes such as control processes with dedicated software systems for monitoring and control highly constrain the resources. Such processes also tend to be event-driven, without a defined top-down proactive specification. For processes of this kind it is possible to use the constraints on resources to greater effect in generation of the initial partial order of activities space.

Process model construction. Based on the enumeration of all potential trees, the Process Model Generator must construct a hypothetical initial process model. At this stage, the demonstration episodes that were captured serve as a guide and/or a reality check in the process model construction. However, the enumerated potential trees may contain other knowledge that is not present in the demonstration artifact, depending on the extent and quality of the domain knowledge. Therefore, this generated process model will describe activities that were not present in the demonstration.

As an example of this, let us continue with the partial order state space example from above. In this situation, the partial order state space has to be transformed into a process model. The state space based on artifact element combinations or resource application sequences contains enough information to suggest some aspects of a process model. These aspects include a set of activities, their type hierarchy, their functional decomposition and partial order of activities execution. In a way, the process model succinctly specifies the distinguishable trees by a set of abstractions more suitable for analysis and execution of the process.

The set of activities can be deduced based on the sets of transitions going into the states in Fig. 2, i.e. sequences of artifact transformations without explicit representation of process activities. The result of this mapping is shown in Fig. 3 in which the rectangles correspond to activities and the color of their border corresponds to the color of transitions in Fig. 2 on which they were based. By using data flow analysis we can construct a partial process model that represents functional decomposition of activities and the constraints on their execution. The partial process model in an Atlas-like process language for this assembly example is shown in Fig. 4. This model is partial because it only represents process activities deduced from legal transformations of input artifacts into a final product. Such a way for initial process generation is possible in problem domains where rigorous well-formedness constraints on legal artifact combinations are known. For instance assembly of physical objects out of available parts is such a problem domain.

Rigorous well-formedness constraints are not available in all process domains. Then it is difficult to generate a partial order state space based on artifact combinations. Other domain knowledge has to be used to form an initial process model in such cases. For instance, the set of tools with well-defined functionality or domain-specific intents can be used to suggest either sequences of tool applications or partial order of sub-intents that can lead to a product.

4.2 Learning mechanisms

The learning mechanisms we considered are supervised neural network learning, reinforcement learning, and evolutionary computation [1]. The hybrid method will leverage the strengths of each of the individual methods. Such process aspects as duration, cost, quality, process activity clustering (to generate or confirm activity decomposition captured by the synthesized process) and guidelines/automation for making control flow choices in the synthesized process are expected to be learned by these mechanisms. To our knowledge, these mechanisms have not been applied to procedural learning previously, we expect it to be a novel experience.

The learning mechanisms in the process synthesis integrator cannot assume sufficient examples of desired behavior exist to learn a process using supervised or statistical approaches. In this case, effective actions are learned by exploring alternatives and their outcomes in the simulator, using reinforcement learning and neuroevolution. The process specifications learned earlier allow simulating the process with some degree of accuracy, and the observation can be transformed into a demonstration episode that allows evaluating and generating feedback to the learners.

In situations where the information consists of discrete state variables and discrete actions, and the state is fully known, reinforcement learning [10] is an effective approach. Through Q-learning, a table of values indicating the expected utility of each action in each state is learned; this table is then used to select appropriate actions. For example, reinforcement learning can be used to diagnose failure reports and select compensatory actions.

In other situations, the state is not fully known, and the state and the actions are described with continuous values. Such situations are difficult for reinforcement learning because it is hard to discretize the space and to identify which utility values need to be changed. However, recurrent neural networks can be constructed through evolutionary learning, and can perform robustly in such situations [6,8]. For example, a recurrent network can monitor sensor and navigational inputs from the missile, and guide it to the target even when its exact position is uncertain. Such a network can be evolved in the simulator, by allowing a population of neural networks to control the process, and reproducing the networks that perform well.

These learning methods are brought together to learn an effective decision policy for the process. The learning uses the domain knowledge in the simulator and in the constructed episodes, as well as in the demonstration episode (observation). The decision policy is initially represented statistically in terms of neural network weights and Q-tables. Using standard techniques for knowledge extraction, this knowledge is then translated into a rule-based description of the process [9,10,11]. In this manner, the project not only results in a practical method for learning a generalized process description for a given domain, but also leads to an important scientific conclusion: understanding of how the different learning algorithms compare, i.e. what kinds of tasks they can each solve well, and what kind of knowledge they learn in the process.

5. Conclusions and Future work

The experience we have with synthesizing assembly processes based on the bill of materials and the final product description is encouraging, yet the vast majority of work still lies ahead. We need to make the process model generator be able to use a more generalized description of the final product that does not directly refer to the artifacts in the bill of materials. The other approaches to generation of the initial process model that do not rely on availability of the rigorous artifact well-formedness specifications must be implemented. The learning mechanisms must be evaluated based on their applicability to capturing the various process aspects described. Finally the whole integrator must be evaluated on real-life process domain knowledge from various process domains, to name a few: mechanical assembly, web-services integration, mechanical object control procedures, software development, crisis responses.

We would like to thank Dr. Misty Nodine of Telcordia Research for discussions, useful comments, and editing, Dr. Donald Baker of Telcordia Research for his help with the Atlas process execution system, and Prof. Risto Miikkulainen of the University of Texas, Austin for discussions and help with the application of reinforcement learning and neuroevolution methods to the process synthesis.

6. References

[1] Stanley, K. and Miikkulainen, R. (2002). Evolution of Neural Networks through Augmenting Topologies. Evolutionary Computation, 10:99--127.

[2] 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), July, 1998, pp 215-249.

[3] 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), August 1998, pp 650-663.

[4] 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), April, 1999.

[5] Alexander L. Wolf and David S. Rosenblum, "A Study in Software Process Data Capture and Analysis", *ICSP 2 - 2nd International Conference on Software Process*, February, 1993, pp. 115—124.

[6] Moriarty, D. E., Schultz, A. C., and Grefenstette, J. J. (1999). Evolutionary Algorithms for Reinforcement Learning. *Journal of Artificial Intelligence Research*, 11:199-229.

[7] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986), Learning Internal Representations by Error Propagation. In Rumelhart, D. E. and McClelland J. M., *Parallel Distributed Processing*. Cambridge, MA: MIT Press.

[8] Stanley, K. and Miikkulainen, R. (2002). Evolution of Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10:99-127.

[9] Stanley, K. and Miikkulainen, R. (2004). Competitive Coevolution through Evolutionary Complexification. *Journal of Artificial Intelligence Research*, 21:63-100.

[10] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

[11] Towell, G. G. and Shavlik, J. W. (1993). The Extraction of Refined Rules from Knowledge-Based Neural Networks. *Machine Learning*, 13:71-101.

[12] Dewayne E. Perry and Alexander L. Wolf. “Foundations for the study of Software Architecture”, *ACM SIGSOFT Software Engineering Notes*, 17:4 (October 1992).

[13] D.Georgakopoulos, H.Schuster, A.Cichocki, and D.Baker. (1999) “Collaboration Management Infrastructure in Crisis Response Situations”, Technical Report CMI-009-99, Microelectronics and Computer Technology Corporation

[14] D.Baker, A.R.Cassandra, H.Schuster, D.Georgakopoulos, and A.Cichocki. (1999) “Providing Customized Process and Situation Awareness in the Collaboration Management Infrastructure”, Proceedings of the 4th IFCIS Conference on Cooperative Information Systems (CoopIS'99)

[15] Dimitrios Georgakopoulos, Hans Schuster, Donald Baker, and Andrzej Cichocki. (2000) “Managing Escalation of Collaboration Processes in Crisis Mitigation Situations”, Proceedings of the 16th International Conference on Data Engineering (ICDE'2000)

[16] Wagner, Thomas A., Garvey, Alan J. and Lesser, Victor R., “Satisficing Evaluation Functions: The Heart of the New Design-to-Criteria Paradigm”, UMass Computer Science Technical Report 1996