Synthesis-Based Loose Programming

Anna-Lena Lamprecht and Stefan Naujokat
Technical University Dortmund
Chair for Programming Systems
44227 Dortmund, Germany
anna-lena.lamprecht@cs.tu-dortmund.de
stefan.naujokat@cs.tu-dortmund.de

Tiziana Margaria
University Potsdam
Chair for Service and Software Engineering
14882 Potsdam, Germany
tiziana.margaria@cs.uni-potsdam.de

Bernhard Steffen
Technical University Dortmund
Chair for Programming Systems
44227 Dortmund, Germany
bernhard.steffen@cs.tu-dortmund.de

Abstract—In this paper we present loose programming, an approach designed to enable process developers to design their application-specific processes in an intuitive style. Key to this approach is the concept of loose specification, a graphical formalism that allows developers to express their processes just by sketching them as kinds of flow graphs without caring about types, precise knowledge about the available process components or the availability of resources. They only have to specify the rough process flow graphically in terms of ontologically defined ‘semantic’ entities. These loose specifications are then concretized to fully executable process code automatically by means of a combination of 1) data-flow analysis, ensuring the availability of the required resources, 2) temporal logic-based process synthesis, resolving type conflicts and taking care of correct component instantiation, and 3) model checking, to ensure global intents and invariants expressed in temporal logic.

I. INTRODUCTION

Process and workflow design or the composition of mashups are high-level tasks that still require significant programming knowledge. Supporting tools typically provide a graphical syntax or a simple scripting language, which simplifies the handling of services and data, but does not free developers from dealing with APIs and type conflicts.

In this paper we present an approach to overcome this situation by providing a loose graphical specification formalism that is supported by formal methodologies. It bridges the remaining gap to executable process models, and thus enables a form of loose programming: If parts of the process model are marked as loosely specified, the data-flow analysis provides information on available and required resources, which is then used by the temporal-logic synthesis algorithm to find sequences of services that are suitable to concretize the loose parts. Model checking is used to watch global process constraints continuously. As a proof of concept we implemented a loose programming extension to our jABC modeling framework [1] which supports the model-based graphical design, execution and management of software processes.

A loose specification is a “specification which features elements of underspecification” [2]. That is, loose specifications allow one to omit detail, either because it not known yet, or to explicitly describe a larger class of possible implementations. Loose programming is based on a similar notion of looseness, however with a different intent: rather than deferring the necessary concretization to later phases of program development, loose programming aims at an environment where such details can be added automatically according to semantic constraints. More concretely, we allow for horizontal and for vertical looseness and composition. The horizontal dimension addresses the process, including its loosely specified parts, and deals with “what we want to do”. The vertical dimension evaluates taxonomies over types and services, allowing for the usage of abstract type and service descriptions in the specifications. For the concretization of the different loosely specified parts of the process we apply synthesis methodology [3], [4] that takes both dimensions into account: Its specification language SLTL includes modalities that describe aspects of relative time and thereby works on the aforementioned horizontal dimension, and taxonomic expressions over types and modules that address the vertical dimension.

The paper is structured as follows. Section II introduces an example scenario that accompanies the remainder of the paper. Our concept of loose specification is then concretized in Section III, before Section IV describes how the combination of loose specification, data-flow analysis, synthesis, and model checking facilitates loose programming. Finally, Section V addresses related work before we conclude in Section VI.

II. EXAMPLE SCENARIO: LATEX SERVICES AND PROCESSES

For the illustration of our methodology, we use an easily comprehensible example throughout this paper. The scenario is woven around services for editing, converting, and printing of \\LaTeX files. Figure 1 shows the available services, along with their input and output data types. Additional structuring of the domain is provided by the classification of types and services in taxonomies, which are simple ontologies that relate entities in terms of is-a relations. Figure 2 shows the service taxonomy that we defined for our example. We did not define abstract type descriptions for this example, which is why we spare to display the type taxonomy. Processes in our example scenario usually start with EditLatex, somehow produce a PDF file from the edited TEX file and output it.
III. LOOSE SPECIFICATION

Usually, the goal of synthesis is to create a system from a given formal specification. But with increasing complexity of the desired system, this formal specification will become almost impossible for anyone to formulate manually. In order to make this task tractable, we introduce loose specification as a mixed textual and graphical formalism.

In our approach to loose programming, the process under development is represented as a control-flow graph with nodes for the services and branches connecting them according to the process flow. Parts of this process model can be marked as loosely specified. Our concept covers the following two dimensions of loose specification and composition:

1) The horizontal dimension addresses the process, including its loosely specified parts, dealing with “what we want to do”. It allows for branches between services to be underspecified. The developer expects the synthesis to come up with reasonable services to be inserted.

2) The vertical dimension evaluates taxonomies over types and services, allowing for the usage of abstract type and service descriptions in the specifications. This enables the developer to insert loosely specified services into the process.

A control-flow graph with loose components provides a comprehensive specification of the process. It can be seen as one large system of constraints that is translatable into a single mu-calculus formula utilizing the characteristic formula [5] or characteristic system of equations [6], to which then mu-calculus synthesis [7] can be applied. The problem would be decidable, but probably not tractable in practice. Thus, we treat loosely specified parts as subproblems that will be solved separately. Figure 3 summarizes the different types of subproblems that can occur (loosely specified branches and services are denoted by dotted outlines). These subproblems are handled as follows:

- A loosely specified branch between two concrete services defines a synthesis problem where the start types are given by the outputs of the service at the source of the branch and the end types by the input of the service at its target, plus the possibility to specify additional constraints. This horizontal synthesis problem can be described by a simple linear-time formula, say Φ.

- A loosely specified service between two concrete branches is treated as an abstract service description, which is evaluated against a service taxonomy in order to find a matching concrete service. This vertical synthesis problem can be described by a simple taxonomic expression, τ.

- Loosely specified branches next to loosely specified services (i.e. combining vertical and horizontal specification) can be treated in two ways:

  1) They are joined into one formula describing the synthesis problem. For instance, if a loosely specified branch with synthesis problem Φ is followed by a loosely specified service with taxonomic expression τ, this combines to the synthesis problem (Φ before τ). An additional subsequent loose branch would lead to (((Φ before τ) before Φ′) before Φ′′), and so on.1

  2) If the combined formula becomes too large to be computed within acceptable time limits (the reason for us to aim at handling subproblems in the first place), the loosely specified service can be instantiated with possible concretizations first, then the loose branches are resolved independently into concrete sequences by the synthesis.

Each of these synthesis problems can be treated locally, i.e. without taking the context of the surrounding process into account, or globally, i.e. with respect to the resources that are available in the global context of the process.

In the local case, the synthesis problems described above are simple, context-independent formulae. If we have, for instance, a loosely specified branch between the services Latex and Print somewhere in the process, the synthesis problem Φ specifies a sequence of services that takes Latex’s output (DVI) as input and finally produces Print’s input (PDF).

1 According to [3], the operator before is defined as follows:

Φ before Ψ =<Φ (¬Ψ) U (Φ A ¬Ψ) A F Ψ>
Treating the synthesis problem globally means to take the resources of the whole process, i.e. all contained data and their types, into account. This requires data-flow analyses on the process model, in the simplest case determining the types and expressions that are available at the different states in the model. With respect to the previous example, this means that at the state of the process that is represented by \texttt{Latex}, not only its outputs are available, but possibly other data that has been produced by preceding services. For instance, a TEX file created by \texttt{EditLatex} might still be available and can also be used within the synthesized sequence.

IV. REALIZATION OF LOOSE PROGRAMMING

The \texttt{jABC} modeling framework [1] supports the model-based graphical design, execution and management of software processes. It provides an intuitive, graphical user interface for dealing with heterogeneous services on a user-centric level. As a proof of concept we implemented our approach to loose programming as an extension to the \texttt{jABC}. In the following we present how our notion of loose specification is combined with synthesis, data-flow analysis and model checking to allow for loose programming within the \texttt{jABC}.

A. Process Synthesis

The synthesis method we use is based on the modal logic SLTL, which combines relative time with taxonomic classifications of types and services [3]. In the following we describe

1) how a synthesis universe is built from the provided domain knowledge,
2) how a modal logic can be used for loose process specification, and
3) what the synthesis algorithm can finally derive from this information.

Domain knowledge can be derived from service meta-data or provided by a domain expert. It is usually not the business of the process developer, who simply works with it via the loose programming framework. The domain knowledge includes basic descriptions of the available services as well as statements about type compatibility and dependencies between services. It can be extended by hierarchically organizing types and services in taxonomies, i.e. simple ontologies that relate entities in terms of is-a relations. With these taxonomies we can classify types and services according to semantic properties. The taxonomies are considered by the synthesis algorithm when evaluating type or service constraints. The originally defined services and types are named concrete, whereas semantic classifications are named abstract.

The synthesis universe constitutes the search space in which the synthesis algorithm looks for solutions to the synthesis problem. It combines the domain knowledge into an abstract representation of all possible solutions. Synthesis universes tend to be very large. They are therefore never constructed explicitly. Rather they are specified by sets of constraints that are evaluated on the fly.

Formally, the synthesis universe is defined as a triple \((T, S_c, \text{Trans})\) where

- \(T\) is a set of concrete and abstract types
- \(S_c\) is a set of concrete services
- \(\text{Trans} = \{(t, s, t')\}\) is a set of transitions where \(t, t' \subseteq T\) and \(s \in S_c\).

Note that the synthesis universe only contains the set of concrete services \((S_c)\), as it contains real executable paths. Abstract services \((S_a)\) are solely used as part of the formula. The set of all services will be denoted as \(S = S_a \cup S_c\). The sets of types \(T_c, T_a\) and \(T\) are defined likewise.

As the definition of the synthesis universe implies, the domain knowledge comprises basically two sets: types \((T)\) and services \((S)\). The set of types that is available in the domain forms the static aspects, i.e. the data-flow facts that are used as atomic propositions by the underlying logic. The set of services represents the dynamic aspects of the domain, which can be used as actions within the SLTL formulae. Each service is characterized by four subsets of \(T\):

- use are the types that must be available before execution of the service (i.e. the input types of the service),
- forbid describes a set of types that must not be available before execution of the service,
- gen is the set of types that are created by the execution of the services (i.e. the output types of the service),
- kill defines those types that are destroyed and therefore removed from the set of types that were available prior to execution of the service.

In our \texttt{Latex} example, the input types as given in Figure 1 constitute the use sets of the services, and the output types the gen sets. For instance, \texttt{EditLatex} generates an output in the BIB format, while \texttt{Print} uses the type PDF. \texttt{Latex2} and \texttt{PdfLatex} have the types BBL and TEX in their use set and gen sets consisting of DVI and PDF, respectively.

With the definitions of use, forbid, gen, and kill, a service \(s \in S_c\) can be defined as a transformation on the power set of types as follows:

\[
s : 2^T \rightarrow 2^T \quad \quad t \mapsto (t \setminus \text{kill}(s)) \cup \text{gen}(s)
\]

For \(s\) to be admissible in a type state \(t \subseteq T\), the following conditions must be met: \(\text{use}(s) \subseteq t\) and \(\text{forbid}(s) \cap t = \emptyset\).

At last, the synthesis universe can be constructed from the service definitions as follows: For each \(t \subseteq T\), a state in the universe is created. The transition \((t, s, t')\) is added to \texttt{Trans} iff \(s\) is admissible in \(t\) and \(t' = (t \setminus \text{kill}(s)) \cup \text{gen}(s)\).

With this perspective, each (finite) path in the synthesis universe represents a possible process. An excerpt from the synthesis universe for our \texttt{Latex} example is shown in figure 4². From a given initial state the synthesis algorithm searches this universe for a path satisfying the given specification.

²Note that for legibility reasons many states and transitions are missing in this figure, especially the reflexive edges that would represent an admissible service that does not change the state (e.g. because its gen types already exist in that state). Furthermore, as there are no kill definitions in our example domain, every path (i.e. process) in the configuration universe is monotone regarding the types, resulting in a partially ordered configuration universe.
So far, we discussed how to incorporate static domain information. As described in Section III, we can have a local or a global view on the available types in the model. In particular, we aim at the identification of start types for the synthesis. In order to have an initial state for the search within the synthesis universe, we need to determine the types that are available prior to execution. As we expect more feasible solutions with the global view, these types are collected via data-flow analysis on the process model. The required Available Types Analysis is an adaption of the common Available Expressions Analysis (cf. [8]). It collects all types that are available prior to the execution of a service, independently from the path the execution has taken before (must-Analysiss).

The synthesis problem is expressed in a logic called SLTL (for Semantic Linear Time Logic) [3], a semantically enriched version of PLTL that is focused on finite paths. The model is given as a path in the previously defined synthesis universe. The syntax of SLTL is defined by the following BNF:

$$\phi ::= true \mid t \mid \neg \phi \mid \phi \land \phi \mid (s_c)\phi \mid G\phi \mid \phi U \phi$$

where $t_c$ and $s_c$ express type and service constraints.

Thus, SLTL combines static, dynamic, and temporal constraints. The static constraints are the taxonomic expressions (boolean connectives) over the types or classes of the type taxonomy. Analogously, the dynamic constraints are the taxonomic expressions over the services or classes of the service taxonomy. The temporal constraints are covered by the modal structure of the logic, suitable to express the ordering constraints.

For the formal semantics of SLTL, let a path be represented as the alternating sequence of type sets and services. A path $p = (t_0, s_1, t_1, s_2, t_2, \ldots, s_k, t_k)$ with $k \in \mathbb{N}_0$, $t_i \subseteq T_c$ and $s_i \in S_c$ satisfies the formula $\Phi$ (written as: $p \models \Phi$) according to the following recursive definition:

- $p \models true$ applies to any path $p$
- $p \models t$ if $t \in TTAX(t_0)$
- $p \models \neg \Phi$ if $p \not\models \Phi$
- $p \models \Phi_1 \land \Phi_2$ if $p \models \Phi_1$ and $p \models \Phi_2$
- $p \models < s > \Phi$ if $k > 0$ and $s \in STAX(s_1)$ and $p_i \models \Phi$
- $p \models G \Phi$ if $\forall i \in \{0, \ldots, k\} : p_i \models \Phi$
- $p \models \Phi_1 U \Phi_2$ if $\exists i \in \{0, \ldots, k\} : \forall j \in \{0, \ldots, i - 1\} : p_j \models \Phi_1$ and $p_i \models \Phi_2$

The abbreviation $p_i$ is defined as:

$$p_i = (t_1, s_{i+1}, \ldots, s_k, t_k)\text{ if } i \in \{0, \ldots, k - 1\}$$
$$p_i = (t_k)\text{ if } i = k$$

The used functions for the evaluation of the taxonomic information, $STAX$ and $TTAX$ are defined as follows:

- $STAX : S_c \rightarrow 2^S$
  $$x \mapsto \{ s \in S \mid s \text{ lies on a a path from the root node to } x \}$$
- $TTAX : 2^{T_c} \rightarrow 2^T$
  $$X \mapsto \{ t \in T \mid \exists x \in X : t \text{ lies on a path from the root node to } x \}$$

In addition to the operators defined above, it is convenient to derive further operators from these basic constructs, such as the boolean disjunction, the eventually operator $F\phi =_{def} true U \phi$, or the weak until operator $\phi WU \psi =_{def} (\phi U \psi) \lor G(\phi)$. 
Coming back to our LaTeX example, we might have a loosely specified branch between the services EditLatex and Print, meaning that the synthesis problem is to find a sequence of services that converts TEX to PDF (cf. Figure 1). The straightforward formulation of this synthesis problem in SLTL is $F(pdf)$. It is also possible to encode further constraints into the synthesis problem, for instance explicitly enforcing the use of a bibliography-aware tool: $F(bibliographic) \land F(pdf)$.

The synthesis algorithm interprets the SLTL formula that expresses the synthesis problem over paths of the synthesis universe, i.e., it searches the synthesis universe for paths that satisfy the given formula. The algorithm is based on a parallel evaluation of the synthesis universe and the formula (for details on the algorithm, cf. [3]). It automatically generates all service compositions that satisfy the given specification (i.e., universe and formula). Its output is the basis for the final assembly of the corresponding process model. If multiple valid service compositions are found, the user will either be queried to choose one, or the system will choose one according to a previously defined cost function (e.g., by length of solution, computation time etc.).

Figure 5 shows some possible processes and synthesis results if we place a loosely specified branch between the services EditLatex and Print. A shortest solution for the unconstrained synthesis problem is the sequence EditLatex; PdfLatex; Print. For the variant that enforces the use of a bibliography, a possible shortest solution is EditLatex; EditBib; BibTex; PdfLatex; Print. The other solutions are longer, but may be inserted if more constraints are defined, or if the user explicitly chooses one of them.

B. Model Checking

The model checker GEAR [9] allows for evaluating static properties of models within the jABC. It copes with formulae in mu-calculus, which e.g., fully comprises LTL [10], a logic designed to reason about properties on a path or linear model.

In the previous section we showed how constraints that are not contained in the domain model can be encoded into the process specification that is given to the synthesis algorithm. This is, however, not always possible, in particular when these constraints are global, overlapping various areas of synthesis. Thus, we apply model checking to monitor the process development globally and continuously, (the manual as well as the automatic insertion of services into the process model). For instance, consider a global constraint (provided by the domain expert as part of the domain knowledge) saying that if bibliographic data is available, it has to be used for producing the final PDF document. That is, PdfLatex2 should be used, whereas the simple PdfLatex must not. Formally, this is expressed as

$$G(bib \Rightarrow (G(\neg PdfLatex) \land F(PdfLatex2)))$$

Now consider this small example: A loosely specified branch between the services EditLatex and Print might be replaced by the service PdfLatex, which is conform with the above constraint. If later on, an additional EditBib service is inserted into the process, the constraint is not fulfilled anymore and the model checker alerts the process developer by marking the service that violates the condition.

In addition to the manual formulation of such global process properties, others can be derived from the synthesis information, for instance allowing for automatic checking of type consistency within a model using the service domain specification. While this is not necessary for process parts that were produced by the synthesis (as it takes care of those dependencies automatically), it can be useful if the process developer changes the synthesized parts later. To achieve this global type monitoring, for every type $t \in T$ a formula of the following structure is generated:

$$\left( \bigwedge_{u \in U(t)} \neg u \right) \bigwedge (\bigvee_{g \in G(t)} g)$$

where $U(t)$ denotes the services that have $t$ as input (use) and $G(t)$ the ones that have $t$ as output (gen).

V. RELATED WORK

The main goal of our loose programming approach is to create a new form of model-based graphical software development based on intuitive loose specifications that can automatically be concretized to running applications by inserting missing detail. We do not know of any approach that strives for a similarly comprehensive goal. However, there are a number of approaches that consider aspects of it, or that provide adequate theoretical background.

The challenge of automatically translating high-level specifications of (remote) service compositions into technical realizations has already been addressed in [11], [12], [13], where the Declarative Service Flow Language (DecSerFlow) [13] is used as a specification language that is grounded in linear-time temporal logic, and different formal methods are applied for the enactment and verification of the actually resulting workflows [11]. The declarative nature of the DecSerFlow specifications is, however, not directly suitable to meet the skills and desires of process developers that are not specially trained in IT and formal methods. Thus, our loose
programming approach aims at putting a stronger focus on conventional procedural structures in the graphical process specifications, in order to provide a specification mechanism that maintains the simple and intuitive structure of established control flow descriptions.

Data-flow analysis [8] and model checking [14] are common formalisms for static analysis and verification. Synthesis and planning are also well-researched areas, but most of the work in this field is more concerned with algorithms and optimization than with application and usability. Those approaches are mostly based upon Pnueli’s and Rosner’s work [15] on the synthesis of reactive systems (e.g. [7]) or on artificial intelligence-based planning, like, e.g., STRIPS [16] [17, pp. 366 ff.], situation calculus [18] [17, pp. 388 ff.] or Hierarchical Task Networks [19] [17, pp. 406 ff.]. There are also approaches to use specification in linear-time logic to guide planning algorithms [20] [21], typically as a means of runtime optimization. Approaches to loose specification also exist (e.g. [22] [23]), but they all aim at programmers with background in formal methods. They are therefore not directly adequate to be applied for our user-oriented loose programming purposes.

VI. CONCLUSION

In this paper we showed how the interplay of data-flow analysis, process synthesis, and model checking supports a very flexible form of loose programming of processes. Incomplete process models with loosely specified components can be automatically completed to be fully executable or translatable to stand-alone code. We believe that the presented technologies have great potential with respect to making highly heterogeneous services accessible to application experts that need to design and manage complex processes. After an adequate domain modeling, including the definition of the semantic rules to be checked by the model checker or to be exploited during model synthesis, application experts should ultimately be able to profitably and efficiently work with a world-wide distributed collection of tools and data, using their own domain language.

For this paper, we chose the \LaTeX{} example, which is quite small but already sufficient to illustrate the essential features of our methodology. However, we have already gained experience in much larger real-world application domains with comprehensive service collections. In particular, we are working on domain-specific service composition in the field of bioinformatics [24]. In a major case study [25], we apply the presented methodology to the EMBOSS suite, a large collection of bioinformatics analysis tools. The corresponding domain comprises a total of over 170 heterogeneous services and numerous taxonomic classifications of the types and services. Based on this domain model biologists are able to loosely program/model complex workflows from the large, heterogeneous, and hence manually intractable EMBOSS collection. Currently, we are extending the scope of this project to incorporate third-party domain information, and therefore to support workflows across various technologies and platforms.

REFERENCES