Formal Specification and Verification of Well-formedness in Business Process Product Lines

Giselle Machado
Computer Science Department
University of Brasília
Brasília, Brazil
Email: gisellegiba@gmail.com

Vander Alves
Computer Science Department
University of Brasília
Brasília, Brazil
Email: valves@cic.unb.br

Rohit Gheyi
Department of Computing and Systems
Federal University of Campina Grande
Campina Grande, Brazil
Email: rohit@dsc.ufcg.edu.br

***Abstract*—Quality assurance is a key challenge in product lines (PLs). Given the exponential growth of product variants as a function of the number of features, ensuring that all products meet given properties is a non-trivial issue. In particular, this holds for well-formedness in PLs. Although this has been explored at some levels of abstraction (e.g., implementation), this remains unexplored for business process PLs developed using a compositional approach. Accordingly, in this paper we report on-going work in formalizing Business Process Product Lines, including the definition of well-formedness rules, formal specification of transformations, and the proof that transformations preserve well-formedness without having to instantiate all variants. Formalization and proofs are provided in the Prototype Verification System, which has a formal specification language and a proof assistant.**

***Index Terms*—Business Process Product Lines, Verification, PVS, Formalization, Well-formedness, Compositional Approach**

I. Introduction

Business processes are a way for an organizational entity to organize work and resources (people, equipment, information, and so forth) to accomplish its aims [1]. These processes specify key activities, roles, and artifacts produced in a specific manner by an organization. Some activities are performed manually and others can be automated by the development of one or more systems. Such processes are essential for achieving business goals and compliance to them is an important measure of organizational maturity [2].

It is often the case that replication of activities or even whole processes occur and failure in identifying such replication results in unnecessary organizational costs regardless of the quality of the underlying software supporting the existing processes [3]. In order to minimize this risk and optimize allocation of organizational resources, it is important first to be aware of this replication and then to handle such commonality and variability. Accordingly, Business Process Product Lines (BPPL) [4], [3] have emerged to leverage Software Product Line concepts and techniques to business processes: a BPPL is a product line (PL) whose products are sets of business process for a particular organization.

Quality assurance is a key challenge in PLs. Given the exponential growth of product variants as a function of the number of features, ensuring that all products meet given properties is a non-trivial issue. In particular, this holds for well-formedness in PLs. Although this has been explored at some levels of abstraction (e.g., implementation), this remains mostly unexplored for BPPL. Although recent work investigates this issue formally for business processes [3], it has only been done for an annotative approach [5] and the proof is not machine-checkable, which may increase the likelihood of error. Annotative approaches have well-known modularity and legibility issues, whereas compositional approaches mitigate these issues and have also been largely used for managing variability in PLs. Further, from the formalization side, a machine-checked proof is sufficiently detailed for replication and certification purposes.

Accordingly, this paper reports on-going work using the Prototype Verification System (PVS) [6] to formally specify and verify well-formedness in BPPL developed using a compositional approach[1]. In particular, the contributions are threefold:

- **Formal Specification** (Section II): the formalization in PVS precisely specifies a language of BPPL, well-formedness properties, and compositional transformations. It focuses on how these transformations contribute to well-formedness.
- **Scalable and General Verification** (Section III): the verification is formal and ensures well-formedness of all products without checking each one individually. It consists of proofs that representative transformations preserve well-formedness in any BPPL.
- **Experience** (Section IV): the paper reports experience in using this proof assistant, which can be useful for similar purposes.

II. Formalization of BPPL

This section presents our formal specification of BPPL. The formalization is based on an existing Haskell implementation [7]. Since Haskell is a functional programming language with limited formalization support, we decided to perform the formalization in PVS, because it contains a sufficiently expressive specification language, e.g., first-order logic, for the purpose of this work. Further, this was also practical because PVS has a functional subset to which Haskell constructs can

[1]The PVS specification and proofs can be found online

be mapped, e.g., Haskell's functions and algebraic data types map into PVS's functions and datatype, respectively.

For readability purposes in this paper, we render some of PVS symbols with their usual mathematical representation. The overall structure of the specification is shown in Figure 1. The arrows indicate the dependency relationships between modules, each of which is a PVS theory (group of related definitions, e.g., functions, data types, theorems). The specification is divided into four PVS theories: 1) *Example* has concrete examples of BPPL (Section II-A); 2) *Language* defines the syntax and well-formedness rules of the BPPL (Section II-A); 3) *Transformations* define transformations over BPPL (Section II-B); 4) *Properties* are theorems on well-formedness of BPPLs when these are manipulated by such transformations (Section III-A).

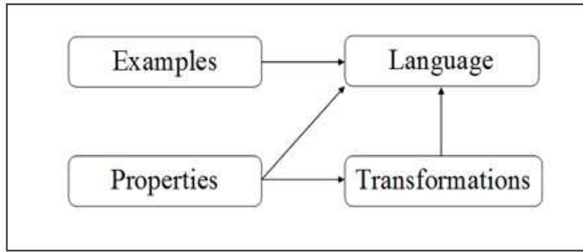

Fig. 1. Overall structure of formal specification of BPPL.

A. Business Process Product Line Language

We now define BPPL's language in PVS, by first specifying its abstract syntax (Section II-A1) and then its well-formedness rules (Section II-A2). We note that BPPL's semantics in this paper is limited to well-formedness properties.

1) Abstract Syntax: it consists of hierarchically related algebraic data types defining key concepts in the BPPL domain such as business process model and business process. Translating it into a PVS specification from the initial Haskell implementation mapped corresponding constructs between these two languages: Haskell's algebraic data types map into PVS's datatype; further, list was mapped to set, since the order of elements was not relevant for specifying the transformations and the properties.

Below is the PVS specification of `BPModel`, the root type in the abstract syntax; it consists of a set of processes of type `BP` (business process). `BPM` is a type constructor and `BPModel?` a type recognizer.

```
BPModel: DATATYPE
BEGIN
 BPM(processes:ℙBP): BPModel?
END BPModel
```

A `BP` is a type which has four fields (an identifier, a type, a set of objects and a set of transitions), each of each is further modeled as a PVS datatype. The following is an example of a BPPL, namely `M`, comprising a single process (Figure 2). Since scope of names in PVS is valid from the point of definition to the end of the specification, the definition has a bottom-up style.

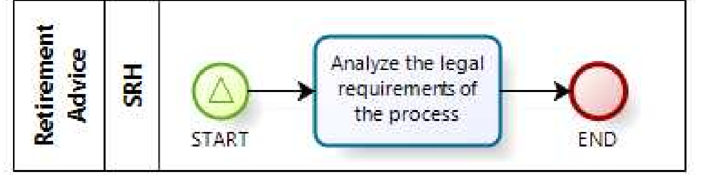

Fig. 2. Example of *Advice*

The process flow objects are `Start`, "*Analyze the legal requirements to the process*" (namely `C`) and `End`. `Start` and `End` are unique objects. `C` is of the type have the `FlowObject` and has four fields: an identifier (`id4`), a type (`Activity`), a set of annotations (`a1`) and a set of parameters (`p1`):

```
id1: idFO
a1: Annotation
p1: Parameter
C: FlowObject = CFO(id1,Activity,a1,p1)
```

The arrows are called transitions. A transition connects two flow objects. The first transition is represented by `t1`, connecting `Start` to `C`. This transition has a condition `c1`. The second transition (second arrow) is `t2`, which connects `C` to `End` and has condition `c2`:

```
c1,c2: Condition
t1: Transition = MkTransition(Start,C,c1)
t2: Transition = MkTransition(C,End,c2)
```

The business process "*Retirement*" (`bp1`) comprises a handle (`id`), a type (`Advice`), a set of objects (`objs`) and a set of transitions (`ts`).

```
id: idBP
objs = {Start, C, End}
ts = {t1, t2}
bp1: BP = BP(id,Advice,objs,ts)
```

Finally, this simple BPPL comprises a value of the `BPModel` type, in this case a set of processes.

```
M: BPModel = BPM({bp1,bp2})
```

Currently, the language does not address other business process constructs such as gateways, swim lanes, and events. Nevertheless, a similar mapping approach employed could be used to address these and is regarded as future work.

2) Well-formed Rules: In the original Haskell specification [7], well-formedness rules were built into the typechecker. These rules and their mapping to PVS are explained in the following. First, a well-formed BPPL, described by the `wfModel` predicate, comprises a set of well-formed business processes:

```
wfModel(p: BPModel): bool =
 ∀ bp: processes(p) | wf(bp)
```

A well-formed product is defined by the `wf` predicate and satisfies the following rules:

```
wf(p: BP): bool =
  WF1(p) ∧ WF2(p) ∧ WF3(p) ∧ WF4(p) ∧
  WF5(p) ∧ WF6(p) ∧ WF7(p) ∧ WF8(p) ∧
  WF9(p)
```

In the following, we illustrate some of these rules. For example, every process must have a beginning and an end, i.e., the `Start` and `End` objects should belong to the set of objects of the `BP` (`WF1`).

```
WF1(p: BP): bool =
  Start ∈ objects(p) ∧ End ∈ objects(p)
```

Any `FlowObject` of `bp` must be reachable from `Start` (`WF5`). To formalize this rule, we first define the `extends` predicate, which relates two identical flow objects or those comprising a transition in the business process:

```
extends(bp: BP, origin: FlowObject)
        (last: FlowObject): bool =
  origin=last ∨
  ∃ t:Transition | t ∈ transitions(bp) ∧
                   startObject(t) = origin ∧
                   endObject(t) = last
```

Next, we consider the reflexive transitive closure of the `extends` relation. In PVS, the inductive definition `extendsClosure` specifies that there is either a direct `extends` relationship between two flow objects or there is a middle object between them. This middle object being reachable from the origin and from which the last object is directly related with `extends`:

```
extendsClosure(bp:BP, origin:FlowObject)
      (last:FlowObject): INDUCTIVE bool =
  origin ∈ objects(bp) ∧
  last ∈ objects(bp) ∧
  (extends(bp,origin)(last) ∨
   ∃ middle: FlowObject |
     middle ∈ objects(bp) ∧
     extendsClosure(bp,origin)(middle) ∧
     extends(bp,middle)(last))
```

To formalize `WF5`, we rely on `extendsClosure` as follows:

```
WF5(p: BP): bool =
   ∀ fo: objects(p) |
     extendsClosure(p,Start)(fo)
```

Other well-formed rules refer to properties of `Start` and `End` objects such as `Start` cannot end a transition, `End` cannot start a transition, and `End` is reachable from any object, and to properties of transitions such as transitions of a business process comprise objects within the process and there is only one transition starting at any given object. Some properties such as the latter are used for simplification purposes and are planned to removed in future work.

B. Transformations

We propose a set of transformations to manage variability in BPPL: by selecting and composing business processes (some of which are advices), specific business processes are built. In the compositional approach investigated, we have formalized two transformations, described in the following sections. These are representative of the remaining transformations and thus formalization of these would follow a similar approach.

1) Select Business Process: `SelectBP` simply selects a given `BP`, identified by an id, from the business processes within a BPPL. The transformation then has two parameters (`bpId` and `pl`) and returns a `BPModel` with only one business process. Further, the transformation requires the following: 1) the returned `BP` must have the same `Id` than `bpId`; 2) the BPPL that is an input parameter (`pl`) must be well-formed, as described next.

```
SelectBP(pl:{p:BPModel | wfModel(p)},
     bpId: {id:Id | ∃ p: processes(pl) |
              pid(p) = id}): BPModel =
```

The transformation `SelectBP` returns a `BPModel` comprising a process from `pl` and having the same identifier as `bpId`.

```
pl WITH [processes := {bp:BP |
   bp ∈ processes(pl) ∧ pid(bp) = bpId}]
```

2) Evaluate After Advice: The `evaluateAfterAdvice` transformation composes a `BP` with another one of type `Advice` resulting in a `BP`. For example, a simple commonality and variability analysis of Figures 3 and 4 yields that the only variable `FlowObject` is "*Analyze the legal requirements of the process*" that is outlined in red in the process "*Retirement*"; the other `FlowObjects` are common to both processes.

To manage such commonality and variability in a compositional approach, in Figure 2, a process of type `Advice` was created. Accordingly, process "*Retirement*" is now generated by means of the transformation `evaluateAfterAdvice`, which composes the process of the type `Advice` with the process "*Resignation*". Three `transitions` are removed and two new `transitions` are created connecting process to the other, as shown in Figure 5.

In PVS transformation, `evaluateAfterAdvice` is specified taking as parameters a business process `bp` and an advice

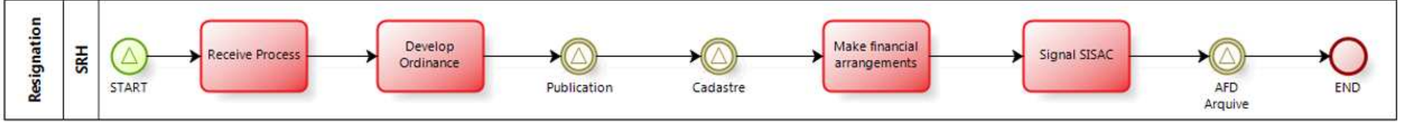

Fig. 3. Process Resignation

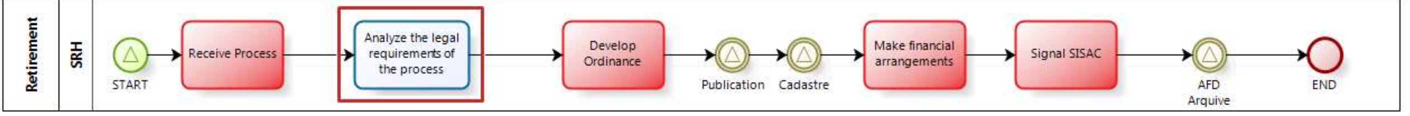

Fig. 4. Process Retirement

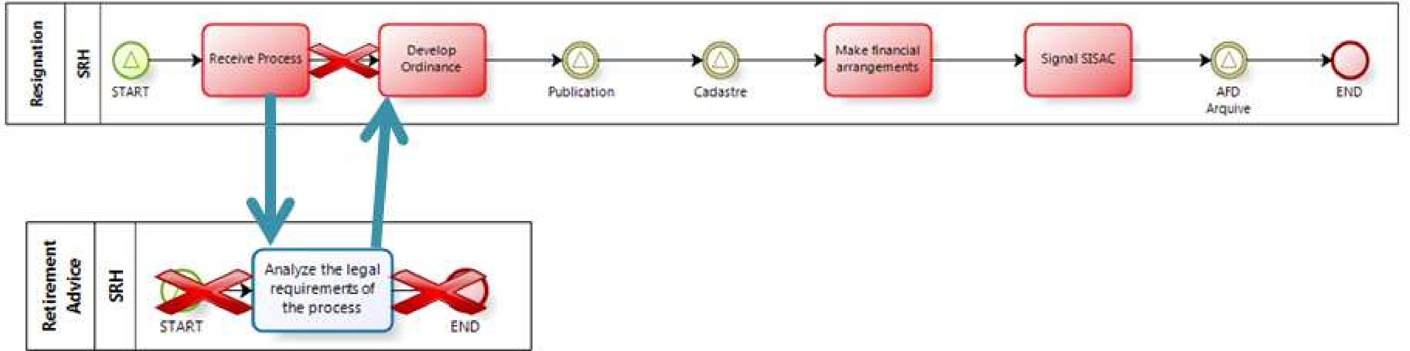

Fig. 5. Example transformation *evaluateAfterAdvice*

`adv`, both of which are well-formed and have disjoints objects. The resulting business process has the union of the objects of `bp` and `adv` and the transitions provided by the `buildT` function, which behaves as shown in Figure 5.

```
evaluateAfterAdvice(adv: {a:BP | wf(a)},
   bp: {b:BP | wf(b) ∧
     objects(b) ∩ objects(adv) = ∅}): BP =
  bp WITH
  [pid := pid(bp), ptype := ptype(bp),
  objects := objects(bp) ∪ FO_adv?(adv),
  transitions := buildT(adv,bp)]
```

In the current specification, we do not address quantification, which is regarded as future work.

III. Verification of well-formedness Properties

Given the exponential growth of product variants in BPPL, ensuring that every product meets certain properties is a non-trivial problem. Seeking to ensure the quality of products BPPL, this study conducted a formal verification of these properties by proving them in the proof assistant PVS. The investment is worthwhile, since these are general properties and thus quantify over infinitely many BPPLs. In this section, we specify a theorem that all transformations must satisfy (Section III-A), and then we show `SelectBP` and `evaluateAfterAdvice` are sound (Section III-B).

A. Theorems

In the product derivation process of compositional BPPL, transformations play a central role by composing business process, some of which are advices, to build a product for any given configuration. A sufficient condition for well-formedness of the resulting product is ensuring that each transformation preserves well-formedness. In this work, we are interested in such transformations regardless of the configuration or BPPL in question.

For example, the `WFM_SBP` theorem states that the `SelectBP(i,pl)` always generate a well-formed BPPL given that `pl` is a well-formed BPPL and `i` refers to a business process in pl, i.e., `SelectBP` preserves well-formedness:

```
WFM_SBP: THEOREM
  ∀ pl: {sp:BPModel | wfModel(sp)},
    bpId: {id:Id | ∃ p: processes(pl) |
                         pid(p) = id} |
      wfModel(SelectBP(pl,bpId))
```

As another example, the following theorem states that `evaluateAfterAdvice(adv,bp)` generates well-formed product given that business process `bp` is well-formed and `adv` refers to an existing advice, i.e., `evaluateAfterAdvice(adv,bp)` also preserves well-formedness:

```
WFM_EVAL: THEOREM
  ∀ adv: {a:BP | Advice?(ptype(a)) ∧
      wf(a) ∧ CPC?(pc(ptype(a)))},
  bp: {b:BP | wf(b) ∧
      objects(b) ∩ objects(adv) = ∅},
  f: FOAnotado?(adv,bp) |
       wf(evaluateAfterAdvice(adv,bp))
```

Theorems for other transformations can be specified similarly.

B. Proofs

In this section, we show that both theorems stated before are sound.

1) Select Business Process: The strategy is to first perform skolemization on the universal quantifier and then expand the sequent according to the definitions of `SelectBusinessProcess` and `wfModel`. Next, instantiate with a variable created in skolemization and obtain two subgoals. The subgoals rely on conditions that are already in the antecedent and consequent, which proves the two subgoals.

2) Evaluate After Advice: The strategy is to break the proof into smaller problems, by creating and relying on auxiliary lemmas during the proof. In particular, this decomposition is accomplished by expanding the `wf` predicate and replicating the context, leading to lemmas (LEMMA_WF1-WF9) corresponding to each of the nine rules as seen in Section II-A2. The lemmas are the same in terms of the context, the variant part being the specification of the well-formed rule. For example, in the following lemma:

```
WFM_WF1: LEMMA
  adv: {a:BP | Advice?(ptype(a)) ∧
      wf(a) ∧ CPC?(pc(ptype(a)))},
  bp: {b:BP | wf(b) ∧
      objects(b) ∩ objects(adv) = ∅},
  f: FOAnotado?(adv,bp) |
       WF1(evaluateAfterAdvice(adv,bp))
```

the last line refers to `WF1`. Other lemmas are defined likewise. We present proof sketches of some of such lemmas. In particular `WF1` establishes that `FlowObjects` (`Start` and `End`) should be in the set of objects `bp`. Given the transformation `evaluateAfterAdvice` (Section II-B), the set of objects of the result of `evaluateAfterAdvice` is composed by the union of objects `bp` with objects of `adv` without the `Start` and `End`. Since `bp` is well-formed, it meets `WF1` and so does the `BP` resulting from `evaluateAfterAdvice`. This proves `LEMMA_WF1`.

As explained in Section II-A2, `WF5` refers to reachability from the `Start` object, i.e., all object are reachable from `Start`. It is known that this rule is valid for `bp` e `adv`, as they are well-formed. To prove `LEMMA_WF5`, we rely on further finer-grained lemmas corresponding to the three groups of objects arising from the partition incurred by the annotated object (joinpoint) matched by the pointcut of the advice as shown in Figure 6. The partition is as follows. The first part is composed of objects in the base process `bp` up to and including the annotated object and is related to lemma `Lema_WF5_01`. The second part comprises all objects in the advice except `Start` and `End` and is related to lemma `Lema_WF5_02`. Finally, the third part is made of objects from `bp` after the annotated object and corresponds to lemma `Lema_WF5_03`. Each such lemma refers to reachability from `Start` (of `bp`) up to the corresponding object partition. `Lemma_WF5_01` follows from well-formedness of `bp`. `Lemma_WF5_02` follows from this, from the definition of the transformation, and from well-formedness of `adv`. Lastly, `Lemma_WF5_03` follows from this, from the definition of the transformation, and from well-formedness of `bp`. In PVS, all such proofs rely on induction, since the definition of reachability is inductive (c.f. Section II-A2). This concludes the proof of `Lema_WF5`.

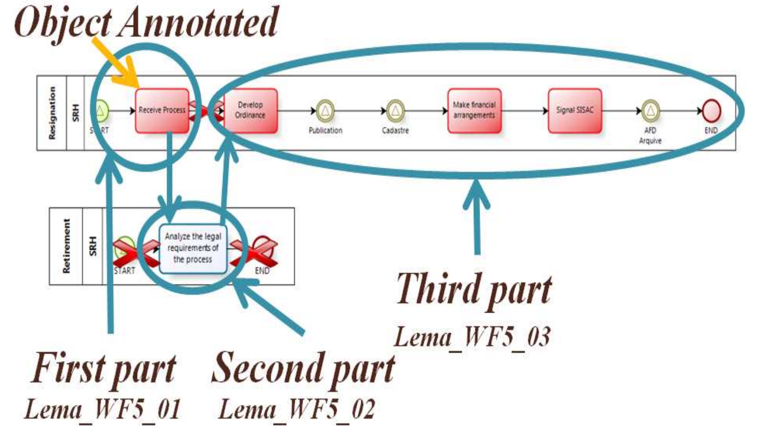

Fig. 6. Partition of objects to prove reachability property under `evaluateAdvice`.

IV. Discussion and Lessons Learned

Since documentation in PVS is limited, there are many details in the specification and proof that are only learned from experience. Indeed, a first lesson was learned about the impact of the order of specification on proofs. For instance, the inductive specification of the function `extendsClosure` (Section II-A2) is better done by placing the base case first instead of the recursive one, the reason being to simplify the corresponding proof by induction.

Further, we experienced that a slight syntactic change in the specification can have an impact in the proofs, in many cases discharging the whole proof. Therefore, it is recommended a proof style relying on different lemmas, minimizing syntactic dependency (e.g., avoiding referencing numbers in the sequent) in the proof commands (e.g., skolemize, instantiate). For example, this strategy was following in proving `WFM_EVAL` in the specification of `wf`. `WF9` as the rule was the last to be specified and the proofs of some theorems had already been made, then the rule `WF9` was placed in the early

specification of wf to this rule appears last in the hour of trial, not to lose the proof already made.

Additionally, during the proof strategy, we normally start by using the smallest granularity commands until a point where the proof can proceed at higher granularity with built-in tactics. In particular, we often aimed at simplifying the formulas embedded with quantification and connectives so as to avoid expanding the definitions, seeking a cleaner sequent structure, which facilitates the understanding and progress of the proof.

One widely used strategy was to first try to prove theorems manually before using the proof assistant. This was instrumental in identifying suitable decomposition strategies to break a theorem into auxiliary lemmas, seeking to guarantee that the theorem can be proved based on the lemma and that each of these is also proved with reduced effort. For example, this strategy was following in proving WFM_EVAL. An addition benefit of this decomposition is to identify reusable lemmas, which otherwise might clutter and slow down the proof process also in terms of performance.

Indeed, the specification and proof efforts are not trivial. Nevertheless, by building up and reusing experience may speed up the process. Moreover, the return-on-investment is desirable because the proofs certify that the properties and general, i.e., they quantify over infinitely many BPPLs. This is different from model checking, whereby the proof effort is smaller (automatic), but it may also take some time and is also not complete.

V. Related Work

Machado et al. [7] characterize variability in business process and then present an approach to manage such a variability. Like our work, the approach employed is compositional, relying on aspects. Unlike our approach, they have not addressed quality assurance in a formal way.

The naive approach of checking well-formedness of all individual programs of a product line is not feasible because of the combinatorial explosion of program variants. To address this problem, Apel et al. [8] devised a type system for a simplified feature-oriented Java-like language. The type system is sound, i.e., it ensures that all programs are type safe. Similar to our work, their solution is general with respect to syntactic property of PL instances; additionally, the target language is a simplified language. Differently, they are concerned with program artifacts, whereas our work addresses a high level artifact (business process) and their proof was carried out manually, whereas ours was carried out semi-automatically with PVS. A similar work [9] proposes an automated approach for verifying safe composition for SPLs with explicit configuration knowledge models.

Schaefer et al. [10] provided a foundation for compositional type checking for delta-oriented product lines (PLs) of Java programs. By combining the analysis results of the delta modules with the PL declaration, they showed that is possible to establish that all the products of the PL are well-typed according to the Java type system. Their approach is transformational and works with PL with object-oriented type checking in Java. Differently, our approach is compositional and focuses on a different level of abstraction (business processes).

Rosa et al. [3] propose a method and tool suite for developing processes based on configurable process model. First they defined the notion of C-iEPC (an extension of the Configurable Event-driven Process Chains, C-EPC) and syntactically correct C-iEPC. Next they provided the definition of C-iEPC and configuration and show an algorithm to individualize C-iEPCs. Finally they proved that the algorithm is able to generate the model of given a configuration correct syntactically. Unlike our compositional approach, this proposal employs an annotative approach with runtime binding mode. This clutter the specification with unnecessary details, hampering understandability. Although formal, their approach employ only manual proof, whereas ours employs the PVS prover.

VI. Conclusions

This paper reports on-going work in formalizing Business Process Product Lines in PVS, including the definition of well-formedness rules. We also encode two transformations and prove them sound (preserve well-formedness rules) with respect to a formal semantics in PVS.

As a future work, we intend to formalize all the remaining transformations of BPPL [7]. Investigating the minimally necessary conditions to preserve the properties of the well-formedness and how these interfere with the transformations is regarded as future work. We further plan to specify semantics of other BPPL constructs, such as gateways, swim lanes, and events, and quantification in the aspect subset to augment the expressivity of the BPPLs considered.

Acknowledgment

We gratefully thank the anonymous referees for useful suggestions.

References

[1] M. Dumas, W. Aalst, and A. Hofstede, *Process-aware information systems: bridging people and software through process technology*. Wiley, 2005.

[2] J. Jeston and J. Nelis, *Business process management: practical guidelines to successful implementations*. Butterworth-Heinemann, 2006.

[3] M. Rosa, M. Dumas, A. Hofstede, and J. Mendling, "Configurable multi-perspective business process models," *Information Systems*, vol. 36, no. 2, pp. 313–340, 2011.

[4] H. Rombach, "Integrated software process and product lines," in *ISPW*, 2005, pp. 83–90.

[5] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE*, 2008, pp. 311–320.

[6] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert, *PVS Language Reference*, 2001.

[7] I. Machado, R. Bonifácio, V. Alves, L. Turnes, and G. Machado, "Managing variability in business processes: an aspect-oriented approach," in *EA at AOSD*, 2011, pp. 25–30.

[8] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Type safety for feature-oriented product lines," *ASE*, vol. 17, pp. 251–300, 2010.

[9] L. Teixeira, P. Borba, and R. Gheyi, "Safe composition of configuration knowledge-based software product lines," in *SBES'11: XXV Brazilian Symposium on Software Engineering*, 2011.

[10] I. Schaefer, L. Bettini, and F. Damiani, "Compositional type-checking for delta-oriented programming," in *AOSD*, 2011, pp. 43–56.