Processing Domain-Specific Modeling Languages: A Case Study in Telephony Services

Fabien Latry, Julien Mercadal, Charles Consel

Processing Domain-Specific Modeling Languages: A Case Study in Telephony Services

Fabien Latry, Julien Mercadal, and Charles Consel
INRIA / LaBRI / ENSEIRB
Department of Telecommunications
351, cours de la Libération
F-33405 Talence, France
{latry,mercadal, consel}@labri.fr

ABSTRACT

The Domain-Specific Language (DSL) approach is being actively studied from both a software engineering viewpoint and a programming language viewpoint. It is being successfully applied to a variety of areas such as banking, graphics and networking. Yet, the concept of a DSL is still very vague, making both its applicability and implementation difficult.

This paper introduces a layered approach to DSLs where (1) domain experts are provided with Domain-Specific Modeling Languages (DSMLs), requiring no programming skills and (2) implementation experts deal with Domain-Specific Programming Languages (DSPLs) that require a programming background but abstracts over the intricacies of underlying technologies.

By separating domain and implementation concerns, we show that our layered DSL approach enables high-level tools to be used to both compile and reason about DSML programs. Compilation and program verification amount to defining high-level generative processes.

We illustrate our approach with the domain of telephony service creation. We introduce a DSML for service creation and demonstrate the ease of compiling DSML programs using the Stratego/XT program transformation environment. Two compilation processes are defined for DSML programs targeting (1) a DSPL, illustrating a high-level compilation process and (2) the TLA+ specification language, exemplifying the verification of domain-specific properties.

1. INTRODUCTION

The Domain-Specific Language (DSL) approach is attracting a lot of attention in software engineering as well as in programming languages. The approach has been successfully applied in a number of areas including banking [1, 11], graphics [13, 15], and networking [20, 25]. Yet, a DSL does not have the same meaning and objectives depending on the community considered. Not surprisingly, these two research communities have been studied the DSL approach mostly independently of each other.

From a software engineering viewpoint, a DSL refers to a modeling language that offers notations and concepts that can be directly manipulated by a domain expert to express a solution as a model [11, 21]. A key purpose of a modeling language is to enable to communicate and share solutions among domain experts. Models can also be used to generate implementations, whether manually or automatically [2, 8, 26].

From a programming language viewpoint, a DSL offers syntactic constructs and semantics that are dedicated to a domain [7, 12]. DSLs developed in the programming language community typically require programming skills, making them less accessible to domain experts. Still, compared to a GPL, a DSL enables programs to be concise and high level. Also, it introduces syntactic and semantics restrictions and extensions that allow critical domain-specific properties to be verified [7].

To reconcile these views, we introduce a layered DSL approach, separating domain and programming concerns. This layered approach consists of Domain-Specific Modeling Languages (DSMLs) and Domain-Specific Programming Languages (DSPLs). A DSML allows solutions to be easily and concisely expressed in domain terms, enabling domain-specific properties to be checked. For a given domain, many DSMLs can be defined, providing domain experts with various visual or textual representations, and various degrees of language expressivity, as depicted in Figure 1. In our approach, DSMLs are implemented in terms of a DSPL. A DSPL is a programming language that serves as an interface between the domain expert and the implementation expert (see Figure 1). That is, a DSPL compiler captures the implementation expertise of a given domain in that it factorizes the know-how to implement a DSPL program in terms of a General-Purpose Languages (GPLs), design patterns, frameworks, software architectures, and such.

By separating domain and programming concerns, our DSL approach stages language processings, enabling specific treatments to be introduced at each layer. In a previous work, Latry *et al.* presented an approach to developing compilers for DSLs, centered around the use of generative programming tools [6]. This work specifically targeted DSPLs and used aspects, annotations, and program specialization to compile the various dimensions of a DSPL. In this approach, compilation and verification of a DSL program were achieved by embedding domain-specific information into a program compiled into a GPL. In doing so, they showed that the compiled program was amenable to a variety of generative programming tools.

By introducing DSMLs, the abstraction level is further raised, bringing up issues and opportunities regarding the compilation of models and the verifications of domain-specific properties.

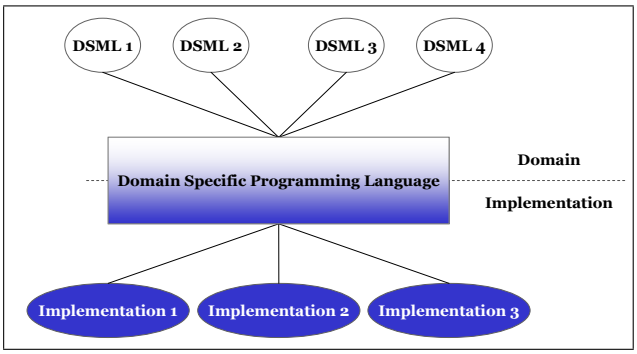

Figure 1: A layered DSL approach

This Paper

This paper presents an approach to compiling and verifying DSMLs. We show that the high-level nature of DSMLs make them amenable to high-level tools that can be used to compile and verify models. In our approach, compilation of DSML models has two main targets: a DSPL and the specification language of a verification tool. In different ways, each of these targets is domain specific: a DSPL offers domain-specific constructs that drastically facilitates the compilation of models, hiding the underlying technologies; a specification language is a dedicated language that consists of specific notations to concisely and formally specify the semantics of a model.

We illustrate our approach with the domain of telephony service creation. We consider a DSML, named Call Processing Language (CPL), that allows non-programmers to visually create telephony services [18, 19]. Furthermore, we focus our compilation process on a DSPL, named Session Processing Language (SPL), which is a programming language dedicated to developing telephony services [4]. Figure 2 displays both the compilation and verification processes that are performed on DSML models. Three transformation passes are performed via XSLT and Stratego/XT: first, an intermediate representation is generated by XSLT from a CPL service, to facilitate further transformations. Then, XT-generated compilers are used to generate SPL programs and TLA+ specifications. An SPL program can be run by the SPL execution environment. While, the TLA+ specification can be verified by the TLC model checker [29].

This variety of processings illustrates the class of new applications of existing tools that is enabled by both the abstraction level of DSMLs and our layered approach.

Outline

This paper is structured as follows. Section 2 introduces our DSL layered approach, based on a DSML layer exemplified by CPL, and a DSPL layer exemplified by SPL. Section 3 presents the compilation of the DSML into two different target languages (SPL and TLA+), and discusses the use of high-level tools such as Stratego/XT to concisely define compilation processes. Section 4 illustrates the ability to use tools to check domain-specific properties. Both sections are illustrated with the domain of the telephony service creation. Section 5 describes some related works and Section 6 concludes.

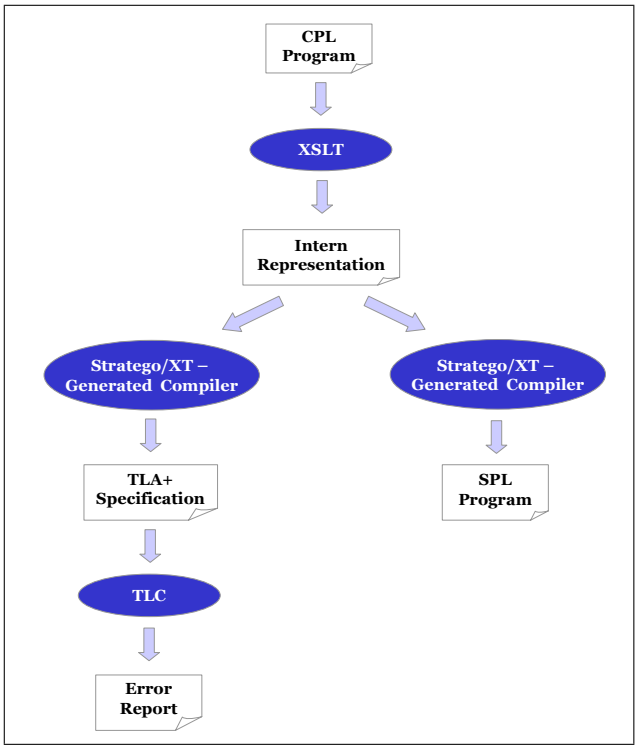

Figure 2: Processing CPL services with high-level existing tools

2. A LAYERED DSL APPROACH

Our layered DSL approach, depicted in Figure 1, consists of introducing two levels in domain-specific languages: DSMLs and DSPLs. A DSML is a modeling language, abstracting over programming concerns. Its mapping to code is facilitated by the domain-specific nature of the target language, namely a DSPL. Because there is a variety of preferences and constraints that can be expressed by domain experts, a variety of DSMLs can be envisioned, offering different visual or textual paradigms, and various degrees of expressivity. The development of these DSMLs is greatly simplified because they can be implemented in terms of a DSPL, making their compilation high level.

Although a DSPL is a programming language, it still abstracts over the underlying technologies of potential target platforms. In this regard, it can be viewed as an interface to the implementation concerns because it exposes the fundamental operations and constructs that need to be implemented when targeting a given platform. This layer improves the ability to re-target a DSPL compiler because a DSPL program can easily be mapped into a target platform; this constrasts with a DSML model whose mapping to code may be arbitrarily difficult. Furthermore, in our approach, targeting a new platform has no impact on the DSMLs.

In the remainder of this section, we examine in detail a

DSML and a DSPL.

2.1 A Domain-Specific Modeling Language – CPL

CPL [18, 19] is an XML-based scripting language for describing and controlling call services. It allows a domain expert to write a telephony service, hiding all the implementation concerns such as distributed system mechanisms and network protocols. A CPL service simply represents a decision tree whose nodes specify predicates and routing decisions to take for processing a call. CPL is designed to be easily parsed and edited by graphical tools. Figure 3 displays a CPL service. In this example, the service handles incoming calls for a user named Bob. When a call occurs, it is forwarded to his current phone. If Bob is busy, then the call is redirected to his voice mail. Otherwise, if he cannot answer (*e.g.,* because he is absent) and the call comes from his boss, then the call is forwarded to his cell phone. The right-hand side of the figure shows the CPL service as a tree-like representation. Because the XML representation of CPL is verbose, and to simplify our presentation, CPL services are considered in their tree-like form hereafter.

As can be noticed in Figure 3, CPL enables a service to be expressed in domain terms; it directly represents the intended decision tree, routing a call. Moreover, its XML coding allows to verify that it conforms with the CPL schema.

2.2 A Domain-Specific Programming Language – SPL

SPL [4] is a programming language dedicated to developing telephony services. It is high level and concise in that it abstracts over a number of underlying technologies like the telephony signaling protocol, the media protocols, and the telephony platform API. Figure 4 presents the SPL version of the CPL service presented in Figure 3. SPL is an event-driven programming language. A program defines handlers for events that cover the complete life-cycle of a telephony session: creation, confirmation, modification, termination. The service logic of our example is only concerned with the creation of a telephony session. As a result, it defines a unique handler for the `INVITE` event (line 6). SPL offers the domain-specific type `response` (line 7) to manipulate call responses. A variable with such type is declared in the `INVITE` handler to store the response of a call forwarding to Bob's phone, when he gets a phone call. Another feature to notice is the SPL syntax corresponding to the protocol response codes (*e.g.,* `/ERROR/CLIENT/BUSY_HERE`), abstracting over raw numbers (lines 8,10). Furthermore, note that a handler can manipulate predefined variables; they are bound to key headers of a call message. SPL offers a lot more domain-specific syntatic and semantic facilities that ease the programming of telephony services[1]. In doing so, SPL bridges the gap between a DSML, like CPL, and its implementation.

3. COMPILING A DOMAIN-SPECIFIC MODELING LANGUAGE

One key benefit of our layered DSL approach is to reduce the gap between a DSML and its implementation, raising the level of the compilation process and enabling the use of high-level tools. We briefly present a program transformation environment, named Stratego/XT, used to rapidly develop compilation processes. Then, we investigate DSML compilation, using of Stratego/XT and studying two different compilation processes.

[1]An extented presentation of SPL is available elsewhere [4].

```
1.  service example {
2.   processing {
3.
4.    dialog {
5.
6.     response incoming INVITE() {
7.      response r = forward 'sip:bob@phone.example.com';
8.      if (r == /ERROR/CLIENT/BUSY_HERE) {
9.        return forward 'sip:bob@voicemail.example.com';
10.     } else if (r == /ERROR) {
11.       if (FROM == 'sip:boss@example.com') {
12.         return forward 'tel:+19175554242';
13.       } else {
14.         return r;
15.       }
16.     }
17.     return r;
18.    }
19.   }
20.  }
21. }
```

Figure 4: Bob's telephony service in SPL

3.1 A Program Transformation Tool – Stratego/XT

Stratego/XT is a framework for creating standalone transformation systems [3, 28]. It consists of Stratego, a language for implementing transformations based on the paradigm of programmable rewriting strategies, and XT, a toolbox for the development of transformation systems. Stratego/XT is used to analyze, manipulate and generate programs. It has been utilized to build a variety of transformation systems including compilers, static analyzers, domain specific optimizers, code generators, etc. In our context, Stratego/XT is used to compile CPL services into both SPL programs and TLA+ specifications (see Figure 2). To do so, the input and output syntax definitions, written in SDF, are used to generate an intermediate component, which is then passed a set of transformation rules in Stratego to produce a standalone compiler. Following this process, we generated two compilers for CPL services targeting SPL and TLA+ [17]. Let us examine these two cases.

3.2 Compiling into a DSPL – SPL

An excerpt of our CPL compiler generating SPL code is displayed in Figure 5[2]. It represents two Stratego transformation rules whose left-hand side corresponds to a CPL construct and right-hand side defines its SPL counterpart. The Stratego rules of our compiler essentially consists of expliciting the decision tree formed by a CPL service. Because the gap between CPL and SPL is reduced, a CPL construct does not require a complicated generation process. Considering our example in Figure 5, the busy test of CPL is simply expanded into a statement forwarding the call, followed by a conditional statement with a test on the response value.

[2]For the sake of readability, Figure 5 shows Stratego rules using the concrete syntax of SPL. However, the actual implementation manipulates the abstract syntax.

```
<?xml version="1.0" encoding="UTF-8"?>
<cpl>
 <incoming>
  <location url="sip:bob@phone.example.com">
   <proxy>
    <busy>
     <location url="sip:bob@voicemail.example.com">
      <proxy />
     </busy>
     <otherwise>
      <address-switch field="origin">
       <address is="sip:boss@example.com">
        <location url="tel:+19175554242">
         <proxy />
        </location>
       </address>
      </address-switch>
     </otherwise>
   </proxy>
  </location>
 </incoming>
</cpl>
```

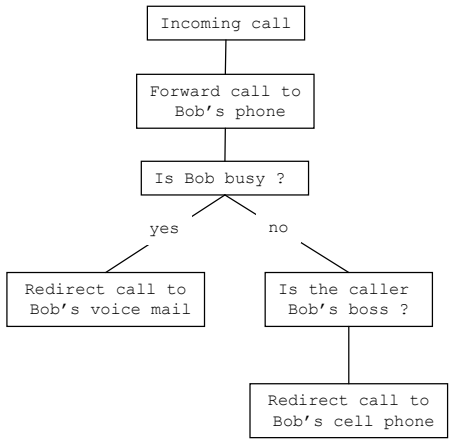

Figure 3: Bob's telephony service in CPL

Another example is the caller test (`FROM`) that is straightforwardly translated into a conditional statement, testing the `FROM` pre-defined variable.

```
RuleRedirectNonTerminal :
    RedirectNonTerminal(callee, BusyTest(stat1*, stat2*)) ->
            |[ response r = forward callee ;
               if ( r == /ERROR/CLIENT/BUSY_HERE ) {
                 stat1*
               } else {
                 stat2*
               } ]|
            where new => r


RuleFROMTest :
    IfElse(FROMTest(caller), stat1*, stat2*) ->
            |[ if (FROM == caller) {
                 stat1*
               } else {
                 stat2*
               } ]|
```

Figure 5: Stratego rules to generate SPL program

Importantly, by targeting SPL a lot of implementation intricacies are hidden, simplifying the compilation process considerably. For example, SPL abstracts over the client-server model used for signaling operations. Specifically, most IP telephony platforms require services to be split into two parts: one to process requests (*e.g.,* `forward`) and one for responses (*i.e.,* the rest of the service). Furthermore, whenever a service requires some state, it needs to be saved prior to invoking the signaling platform and restored for the part of the service processing the response. By targeting SPL, the compiler focuses on the domain aspects of CPL; the implementation concerns are addressed by the SPL compiler.

3.3 Compiling into a Specification Language – TLA+

Compiling CPL services into a TLA+ specification amounts to defining an abstraction that models aspects of CPL for which verification is needed. Specifically, our abstraction consists of modeling the predicates used to determine routing decisions included in a service. For example, a CPL service typically tests calendar events. A cascade of such tests may end up being infeasible. Figure 6 displays an example of a Stratego rule, `RuleTimeDay`, that defines an abstraction of calendar tests. For convenience, this abstraction assumes that a day is represented as a three-element record, consisting of its name, its number in the year, and the name of its month. The abstract version of a service computes a set of such records. A predicate on days leads to computing two new sets: one for the truth branch, including the days verifying the predicate, and one for the false branch, with a complementary set. More specifically, the fragment in Figure 6 defines the Stratego rules enabling to generate TLA+ formulas for a predicate that tests the day of a call. Let us further examine the verification opportunities offered by DSMLs.

4. VERIFYING A DSML – CPL

Checking domain-specific properties is greatly facilitated by both the domain-specific and high-level nature of a DSML. As such an abstract version of a model can be derived more directly. We first briefly present the tool chosen to verify CPL models, considering the kind of computations involved in a telephony service logic. Then, we discuss the domain-specific properties considered for CPL, illustrating them with two erroneous services and an error report from the verification tool.

4.1 A Verification Tool

TLA+ [17] is a high-level specification language used to specify and check the correctness of systems. It is a formal specification language based on Temporal Logic of Actions (TLA) [16], and TLC [29], a model checker for TLA+ specifications. TLA is a simple logic for describing and reasoning about systems. It provides a uniform way of specifying algorithms and their correctness properties. TLC is an on-the-fly model checker for debugging TLA+ specifications.

4.2 Domain-Specific Properties

```
RuleTimeDay :
   IfElse(DayTest(String(day)),
      [Action(Id(thenNext),
          Conjunction([Eq(Id(id3), String(str1)), rest1*])), action1*],
      [Action(Id(elseNext),
          Conjunction([Eq(Id(id4), String(str2)), rest2*])), action2*]) ->
          |[ ifThen  ≜
                        ∧ currentNode = <double-quote> str
                        ∧ currentNode' = str1
                        ∧ date' = {t ∈ date : t.day = day}
                        ∧ UNCHANGED  ⟨sigActions,addrTest⟩
             ifElse  ≜
                        ∧ currentNode = <double-quote> str
                        ∧ currentNode' = str2
                        ∧ date' = {t ∈ date : t.day ≠ day}
                        ∧ UNCHANGED  ⟨sigActions,addrTest⟩
             thenNext ≜
                        ∧ id3 = str1
                        ∧ rest1*
             action1*
             elseNext ≜
                        ∧ id4 = str2
                        ∧ rest2*
             action2* ]|
        where new => ifThen ; new => ifElse ; new => str
        ; rules ( Next :+ Action(Id("Next"), Disjunction(exp*)) ->
            |[ Next  ≜
                        ∨ exp*
                        ∨ Id(ifThen)
                        ∨ Id(ifElse) ]| )
```

Figure 6: Stratego rules to generate TLA+ specification

Because the semantics of a DSML solely involves domain-specific aspects, reasoning about a model is only concerned with domain-specific properties. In the case of CPL, examples of properties include: no call loss, no duplicate redirect, and no infeasible path. These properties are expressed as formulas in Figure 7. They must be verified by the model checker for every telephony service. Let us now examine these formulas in detail.

$$AtLeastOneSigAction \triangleq currentNode = \text{“End”} \Rightarrow Len(sigActions) \neq 0$$

$$NoTwiceRedirectToTheSameURI \triangleq \Box(\forall\, n \in 1\,..\,Len(sigActions) : \forall\, m \in n+1\,..\,Len(sigActions) : sigActions[n] \neq \text{“Continuation”} \Rightarrow sigActions[n] \neq sigActions[m])$$

$$Consistency \triangleq \wedge\, \Box(\exists\, x \in Addresses : \forall\, n \in 1\,..\,Len(addrTest) : x \in addrTest[n]) \wedge\, \Box(date \neq \{\})$$

Figure 7: Properties to verify in telephony services

4.2.1 *No call loss.*

AtLeastOneSigAction is a key property, well-identified by telephony experts [19]. Its goal is to prevent having services that lose calls because not every execution path contain at least one signaling action (*e.g.,* forward and redirect). Specifically, the formula specifies that at an end node, a signaling action must have occurred.

4.2.2 *No duplicate redirect.*

NoTwiceRedirectToTheSameURI ensures that an execution path does not contain more than one redirection to the same address. In doing so, not only are unnecessary operations detected, but the service execution time is kept to a minimum. This property requires the introduction of a signaling trace to formulate that no two signaling actions should be equal. This formula should always be true, as noted by '□'.

4.2.3 *No infeasible path.*

Consistency ensures that a cascade of tests is correct in that it is feasible. Two kinds of tests are considered: (1) addresses, that is, caller identifiers and (2) dates. Consider an example of the former kind, displayed in Figure 8. This service is erroneous in that, when a call occurs, if the sender's address contains Bob (node 2), then the call is forwarded to Bob's cell phone (node 3). Otherwise, if the sender's address is `sip:boss@example.com` (node 5), then the call is redirected to his voice mail (node 6). Nevertheless, this case can never happen because it is subsumed by the first test (node 2). Such a cascade of address tests is common in realistic services, creating a need to verify its feasibility.

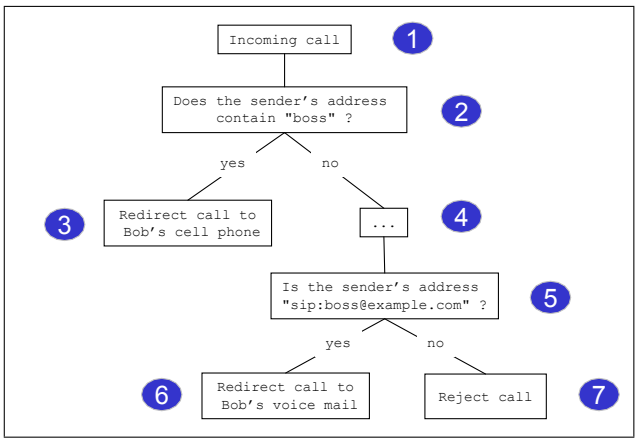

Figure 8: Erroneous CPL service (sender's address)

The second part of the *Consistency* property guarantees that a cascade of tests on calendar events are feasible. An example of an unfeasible execution path is displayed in Figure 9. This service treats calls differently depending on the day of the week, that is, whether a call occurs on Tuesday. If so, it tests whether the call occurs between 07/12 and 07/17. However, there is no Tuesday within that period of time, therefore this last execution path is not feasible.

Because of their decision-tree nature, telephony services include cascades of tests that involve predicates on a variety of aspects. The above examples demonstrate the interest of verifying that a service does not include infeasible paths.

To complete our presentation, we show in Figure 10 an error report produced by TLC, when given to verify the CPL service displayed in Figure 9 that contains an infeasible cascade of tests on dates. In this situation, TLC specifies which property cannot be verified by displaying the service line number where the invariant is violated, namely ($date \neq \{\}$). Then, it generates a counterexample, displaying the sequence of states that leads to the one where the invariant is violated (State 4, with ($date = \{\}$)).

5. RELATED WORK

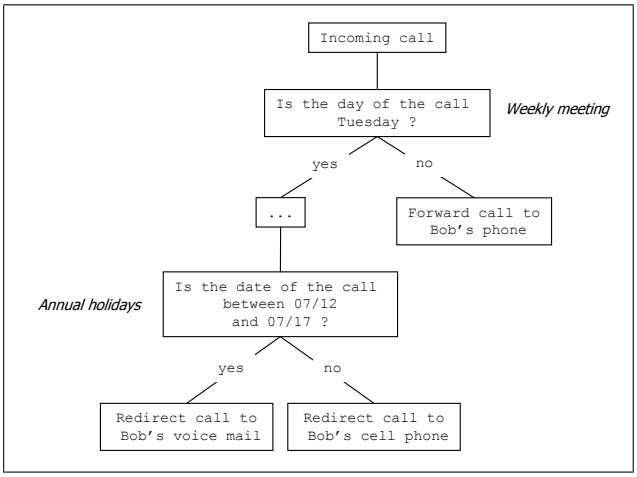

Figure 9: Erroneous CPL service (date of the call)

```
TLC Version 2.0 of January 16, 2006
Model-checking
...
Finished computing initial states: 1 distinct state generated.
Error: Invariant line 143, col 23 to line 143, col 35 of module CPL is violated.

The behavior up to this point is:
     STATE 1: <Initial predicate>
              ∧ addrTest = ⟨ ⟩
              ∧ currentNode = "Incoming"
              ∧ sigActions = ⟨ ⟩
              ∧ date = {[day ↦ "sun", dayNum ↦ 1, month ↦ "January"],
                        ...
                        [day ↦ "sun", dayNum ↦ 365, month ↦ "December"] }

     STATE 2: <Action line 51, col 9 to line 53, col 60 of module CPL>
              ∧ addrTest = ⟨ ⟩
              ∧ currentNode = "WeeklyMeeting"
              ∧ sigActions = ⟨ ⟩
              ∧ date = {[day ↦ "sun", dayNum ↦ 1, month ↦ "January"],
                        ...
                        [day ↦ "sun", dayNum ↦ 365, month ↦ "December"] }

     STATE 3: <Action line 98, col 9 to line 101, col 54 of module CPL>
              ∧ addrTest = ⟨ ⟩
              ∧ currentNode = "AnnualHolidays"
              ∧ sigActions = ⟨ ⟩
              ∧ date = {[day ↦ "tue", dayNum ↦ 3, month ↦ "January"],
                        ...
                        [day ↦ "tue", dayNum ↦ 192, month ↦ "July"],
                        [day ↦ "tue", dayNum ↦ 199, month ↦ "July"],
                        ...
                        [day ↦ "tue", dayNum ↦ 360, month ↦ "December"] }

     STATE 4: <Action line 110, col 9 to line 113, col 55 of module CPL>
              ∧ addrTest = ⟨ ⟩
              ∧ currentNode = "RedirectToBobVoiceMail"
              ∧ sigActions = ⟨ ⟩
              ∧ date = {}

5 states generated, 5 distinct states found, 2 states left on queue.
The depth of the complete state graph search is 4.
```

Figure 10: Error report from TLC

The DSL approach is being actively studied because of its potential to greatly improve software development. As a result, a lot of approaches and tools have been proposed to develop DSLs, addressing both their design and implementation.

Concerning the design, a number of frameworks and development environments are available such as Metacase [27], AMMA (Atlas Model Management Architecture) [5], and Microsoft's initiative on Software Factories [14]. These approaches consist of a high-level and rich toolkit for creating DSLs. It offers powerful graphical environments, to design visual languages, high-level mechanisms, to constrain the composition of language constructs, and a high level of automation, to generate tools required by a new DSL.

Nevertheless, these tools offer limited support for code generation. More specifically, DSLs often introduce a large gap between models and implementations, requiring the intervention of implementation experts to cover all aspects of code generation. The layered solution, presented in this paper, is complementary to DSL development environments: it provides a staged approach that makes the code generation process amenable to program generation and verification tools.

Concerning the implementation of DSLs and their processing, numerous approaches have emerged. Following the classification of model transformation approaches introduced by Czarnecki [10], we identify four main strategies to implement DSLs: code generation, semantic-based compiler generation, model-based transformation, and XML-based transformation.

5.1 Code generation.

Developing a DSL-specific code generator is the main alternative to our approach. It consists of a direct compilation from a DSL to some code. This solution, promoted by the model-driven development approach, presents a number of disadvantages. Firstly, the DSL compiler is often developed as a monolithic program, interweaving implementation and domain concerns. This results in a costly and complicated software development that exposes few opportunities for reuse. Because of the entanglement of domain and implementation aspects, domain experts cannot introduce new verifications without an extended knowledge of the code generator. This situation makes it difficult for a DSL to evolve with the requirements of the domain expert.

In our approach, the code generation process is split into two phases: compiling a DSML and compiling a DSPL. The DSML compiler solely concentrates on high-level domain-specific computations. Verifications can be introduced at this stage without being concerned with implementation issues. The DSPL compiler focuses on implementation concerns, handling low-level details (*e.g.,* platforms, framworks and protocols).

5.2 Semantic-based compiler generation.

The goal of this approach is to generate a compiler from a high-level and formal specification of a language. Denotational semantics is a prime example of this approach; it has been used in a number of compiler generators [24].

Semantic-based compiler generation has mainly been used to generate compilers for general-purpose languages. In this context, this approach has had to compete with hand-crafted compilers, without much success. As a result, little research is still done on this topic nowadays.

5.3 Model-based transformation.

This approach assumes that a DSL is a high-level language and thus is well-suited for model transformation tools like KM3 and ATL [5]. The idea is to define a metamodel for such a DSL and to apply transformations to translate it into another form. In this regard, model-based transformation is complementary to our approach in that it offers additional tools to manipulate DSML models and to map them to verification tools.

5.4 XML-based transformation.

The goal of this approach is to use XML transformers to compile a DSML model into a DSPL program. To do so, the source program is assumed to be represented in XML. Tools like XPath [22] and XLST [23] can be used for this purpose. Nevertheless, some languages may not fit well with XML and lead to the development of cumbersome transformation processes. This is due to the fact that XML tools are ultimately dedicated to data conversion, not program transformation.

As can be noticed, our DSL layered approach is complementary to existing approaches. By introducing both DSMLs and DSLs, it allows language processing to be staged, making a number of tools usable. It is also complementary to rich graphical environment that deals with the design of visual languages.

In our case study, Stratego is used to compile a CPL service into a SPL program and a TLA+ specification; the latter is then processed by the TLC model checker. Noteworthy, TXL (Tree Transformation Language [9]) could have been used instead of Stratego for compilation purposes. Furthermore, other formal tools based on set theory could have achieved our verifications.

6. CONCLUSIONS

This paper presents an approach to compiling and verifying DSMLs. This approach is supported by a layered view of DSLs where DSMLs are concerned with domain aspects and DSPLs are an interface between domain experts and implementation experts. This layering enables DSML compilation to focus on domain aspects, deferring the intricacies of target platforms to the DSPL compiler. In doing so, the development of DSMLs is facilitated by making their compilation more amenable to high-level program transformation tools. Furthermore, DSML compilers become independant of a given platform.

Because the semantics of a DSML only involves domain-specific computations, such language can be reasoned about in a more direct way. As a result, aspects relevant to verifications can be easily extracted from models written in a DSML. Translating such models into the specification language of a verification tools becomes accessible to high-level program transformation tools. The domain-specific nature of DSMLs make properties to be checked more attainable by existing verification tools.

We have validated our approach in the domain of telephony services. To do so, we considered a DSML called CPL, that offers high-level notations to create telephony services. We also used a DSPL named SPL to interface between the domain and the implementation concerns. SPL is a DSL that abstracts over underlying telecommunication technologies but requires programming expertise. These two languages have allowed us to define a CPL compiler targeting SPL. This compiler has been developed using Stratego/XT, illustrating the high-level nature of the transformation process. We also used this tool to compile CPL into the TLA+ specification language. By doing so, we were able to check domain-specific properties using the TLC model checker. This is another example that demonstrates how much DSMLs can leverage on existing tools for compilation and verification.

7. REFERENCES

[1] B. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In *Proceedings of the ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice*, pages 6–13, April 1995.

[2] S. Beydeda and V. Gruhn. *Model-Driven Software Development*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: components for transformation systems. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation (PEPM'06)*, pages 95–99, New York, NY, USA, 2006. ACM Press.

[4] L. Burgy, C. Consel, F. Latry, J. Lawall, L. Réveillère, and N. Palix. Language Technology for Internet-Telephony Service Creation. In *Proceedings of the IEEE International Conference on Communications (ICC'06)*, Istanbul, Turkey, 2006.

[5] J. Bzivin, G. Hillairet, F. Jouault, I. Kurtev, and W. Piers. Bridging the ms/dsl tools and the eclipse modeling framework. In *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, San Diego, California, USA, 2005.

[6] C. Consel, F. Latry, L. Réveillère, and P. Cointe. A Generative Programming Approach to Developing DSL Compilers . In R. Gluck and M. Lowry, editors, *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *Lecture Notes in Computer Science*, pages 29–46, Tallinn, Estonia, September 2005. Springer-Verlag.

[7] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming (PLILP'98)*, volume 1490 of *Lecture Notes in Computer Science*, pages 170–194, Pisa, Italy, September 1998.

[8] S. Cook. Domain-Specific Modeling and Model Driven Architecture. In *The MDA Journal: Model Driven Architecture Straight from the Masters*, chapter 3. D. Frankel and J. Parodi edition, December 2004.

[9] J.R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source transformation in software engineering using the txl transformation system. *Information & Software Technology, Special Issue on Source Code Analysis and Manipulation*, 44(13):827–837, 2002.

[10] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, USA, 2003.

[11] A. van Deursen. Domain-Specific Languages versus Object-Oriented Frameworks: A Financial Engineering Case Study. In *Proceedings of Smalltalk and Java in Industry and Academia (STJA'97)*, pages 35–39. Ilmenau Technical University, 1997.

[12] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[13] Conal E. An Embedded Modeling Language Approach to Interactive 3D and Multimedia Animation. *Software Engineering*, 25(3):291–308, 1999.

[14] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[15] S.N. Kamin and D. Hyatt. A Special-Purpose Language for Picture-Drawing. In *Proceedings of the Conference on Domain-Specific Languages*, pages 297–310, Berkeley, CA, USA, 1997. USENIX.

[16] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[17] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[18] J. Lennox. *Services for Internet Telephony*. PhD thesis, Columbia University, January 2004.

[19] J. Lennox and H. Schulzrinne. Call Processing Language Framework and Requirements. Request For Comments (RFC) 2824, The Internet Engineering Task Force (IETF), May 2000.

[20] P.J McCann and S. Chandra. Packet types: abstract specification of network protocol messages. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'00)*, pages 321–333, New York, NY, USA, 2000. ACM Press.

[21] P. Murray-Rust. Chemical Markup Language (CML). *World Wide Web Journal*, 2(4):135–147, 1997.

[22] W3C Recommendation. XML Path Language Version 1.0. http://www.w3.org/TR/xpath, November 1999.

[23] W3C Recommendation. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, November 1999.

[24] D. A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[25] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, 1998.

[26] J. P. Tolvanen. Domain-specific modeling for full code generation. http://www.methodsandtools.com/archive/archive.php?id=26, 2004.

[27] J. P. Tolvanen. Metaedit+: domain-specific modeling for full code generation demonstrated [gpce]. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '04)*, pages 39–40, New York, NY, USA, 2004. ACM Press.

[28] E. Visser. Meta-Programming with Concrete Object Syntax. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[29] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, pages 54–66, London, UK, 1999. Springer-Verlag.