Implementing Modular Domain Specific Languages and Analyses

Daniel Ratiu
fortiss gmbh
Munich, Germany
ratiu@fortiss.org

Markus Voelter
itemis/independent
Stuttgart, Germany
voelter@acm.org

Zaur Molotnikov
fortiss gmbh
Munich, Germany
molotnikov@in.tum.de

Bernhard Schaetz
fortiss gmbh
Munich, Germany
schaetz@fortiss.org

ABSTRACT

Domain specific languages allow users to directly express domain concepts in their programs and thereby eliminate the accidental complexity resulting from implementation details irrelevant to the domain. Cleaner programs, written in DSLs are much easier to analyze formally. However, domain specific analyses need to be implemented over and over again for each new domain specific language. In this paper we show that the use of language engineering techniques for modularizing languages can drastically improve on this situation. Language fragments (aka. language modules) together with a set of analyses defined for them can be reused between different DSLs, making the implementation of analyses significantly easier. This paper presents our approach for using the Meta-Programming System to implement domain specific languages and analyses both as extensions of C and in the domain of intelligent buildings. The main lesson learned is that modularization at language and analysis level allows rapid instantiation of advanced DSLs and their corresponding analyses.

Keywords

Domain Specific Languages, Validation and Verification

1. INTRODUCTION

Domain-specific languages (DSLs) are languages that are focussed on a particular domain. By making assumptions about that domain and encoding these assumptions in the language itself, a program expressed with a suitable DSL is more concise than a program that expresses the same behavior, but is written in a general-purpose language. A program written in a DSL is also more easily analyzable with regards to the domain, because domain semantics are directly expressed and do not have to be reverse-engineered from a low level implementation. A DSL has exactly the right degrees of freedom, but not more. As an example, consider a DSL that optimizes energy consumption in an intelligent building. A rule could express that *if temperature is higher than 30 degrees centigrade then start the air conditioner*. Identifying those air conditioners that are not controlled by any rule at all is trivial by making sure that each air conditioner is referenced from at least one rule. Doing the same for a C program that controls the air conditioners would be more difficult due to the encoding of the application logic in the low level language.

However, there are also analyses that cannot be easily performed by "just looking" at a program written in a DSL that is suitable for a particular application domain. Coming back to our energy management example from above, the detection of non-controlled air conditioners becomes much more involved if these devices occur in some rules but if the guard conditions for these rules will never become true. This is a typical example of the satisfiability problem, namely checking if a Boolean formula can be true or not. There are many existing tools that can be used to perform such analyses. For each analysis tool we need to transform the DSL program into its input formalism. If we were to analyse different DSLs with different tools, then many such transformations have to be implemented (Figure 1). To simplify this approach, we modularize DSLs along two orthogonal dimensions.

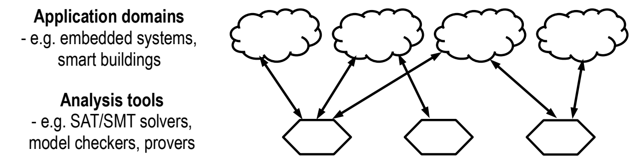

Application domains
- e.g. embedded systems, smart buildings
Analysis tools
- e.g. SAT/SMT solvers, model checkers, provers

Figure 1: More than one analysis tool may be suitable for analyzing programs written in a particular DSL, and/or different DSLs may be analyzed with the same analysis tool.

Language Layering. The *application level* is specifically optimized for a particular application domain, its abstractions, notations and user preferences. There is effectively an *unlimited* set of application domains and associated lan-

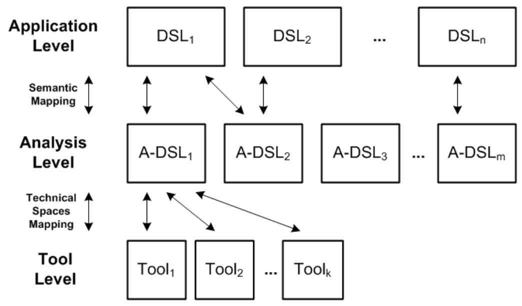


Figure 2: Language layering concerns the separation of the formalism for expressing input suitable for certain analyses from the particular tool used to perform the analyses.

guages. In contrast, the *analysis level* is optimized to be efficiently analyzable (and it may not be directly accessible to application domain users). In practice there is a relatively small set of established formal verification tools that are based on different classes of formalisms. We propose a separation between the language to encode the input for a class of specification formalisms and the particular tools which implement these formalisms (Figure 2). Different tools can be used to perform the actual analyses on the programs expressed with an *analysis level DSL*. The separation of the semantic mapping (from application domain DSL to the analysis DSL(s) and back) from the integration of the actual analysis tool has the benefit that different tools with different non-functional characteristics (performance, license) can be plugged in without changing the semantic mapping. The often less-than-elegant integration of a particular tool (relying on text generation, API bridges, parsing of output) is well encapsulated, effectively forming an anti-corruption layer [3]. The technical integration of a new or additional tool may require a significant amount of work (generation of the input to the tool, interacting with the tool, interpreting the tool results). But these tasks are domain-independent, required to be done only once for each tool, and typically not algorithmically very complex.

Language Modularization. A DSL can often be decomposed into several conceptually distinct sub-languages. Each sub-language is a unit of modularity and in this paper we use interchangeably the terms language module or sub-language. Some of these language modules are amenable to analysis. In the case where a language module is shared between an application level DSL and analysis level DSL, the mapping from an application level DSL to one or more analysis level DSLs is trivial. For example, as we show in this paper, different DSLs may use a sub-language that comprises logical expressions and access to typed variables. Obvious analyses relevant for such expressions include checking whether these expressions cover the whole spectrum of the input values (completeness) and if there are expressions that can evaluate to true at the same time (consistency). By implementing these analyses for a shared language module, the analyses are made available to all the DSLs that reuse this module.

Contributions of this paper. In this paper we show how language layering and modularization enables an easy and reusable implementation of domain specific analyses. We discuss how generic analyses based on satisfiability-modulo theories can be instantiated for two completely different application level DSLs. We show how commonalities between these DSLs can be factored out, simplifying the analyses implementation and making it reusable between different application level DSLs.

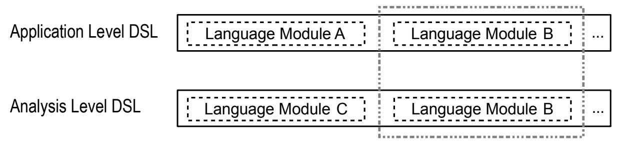


Figure 3: Language modularization refers to the case where several languages reuse a common language module. By defining analyses for the shared language module, the analyses become available for all the languages that use the module.

2. LANGUAGE MODULARITY IN MBEDDR

Over the last couple of years there has been significant progress in the area of language and IDE modularization and composition. In general, modularization is an accepted technique to break down large and complex problems into smaller and more tractable ones. This way, modularization can help to reduce the overall implementation effort and increase quality by reusing already tested parts of a program. The same argument can be made for languages, where reuse and composition has to address syntax, type systems and semantics. In [9, 2, 7] various classification schemes for language reuse and composition have been proposed. While they differ in detail, they all include the following approaches. Language *extension* adds additional constructs to an existing language. Language *restriction* removes some concepts from an existing language. Language *embedding* joins two previously unrelated languages. In all three cases, the original language(s) are not modified invasively, retaining modularity.

2.1 Language Workbenches and MPS

Language workbenches are tools that make language and IDE development, modularization and composition feasible. There are two fundamentally different approaches for language implementation as realized by language workbenches. *Parser-based* systems make use of grammars that specify the textual structure of legal programs expressed in some language l. A parser is generated from that grammar which transforms programs expressed in l into a data structure that contains the information expressed by a program, but gets rid of the textual concrete syntax. This data structure is called an abstract syntax tree (AST). All downstream processing (analyses, transformation) is performed on the AST. *Projectional* systems work without grammars and parsers: as a user edits a program, the AST is modified directly and the program's textual (or other) syntax is merely a projection. While parser-based systems support language modularization and composition to some degree, it is easy to do and well supported in projectional systems. JetBrains MPS (*http://jetbrains.com/mps*) is an example of a projectional language workbench. Its support for language modularization and composition is detailed in [9].

2.2 The mbeddr Stack

mbeddr (*http://mbeddr.com*) is an extensible set of languages for embedded software development based on C. It is discussed in more detail in [10]. mbeddr is implemented based on JetBrains MPS and it exploits its capabilities for language modularization and composition to the benefit of the embedded software developer. At its core, mbeddr supports the incremental, modular domain-specific extension of C. mbeddr also supports language restriction, in order to create subsets of existing languages.

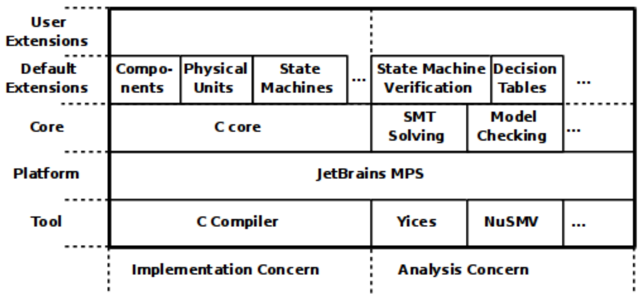


Figure 4: mbeddr at a glance

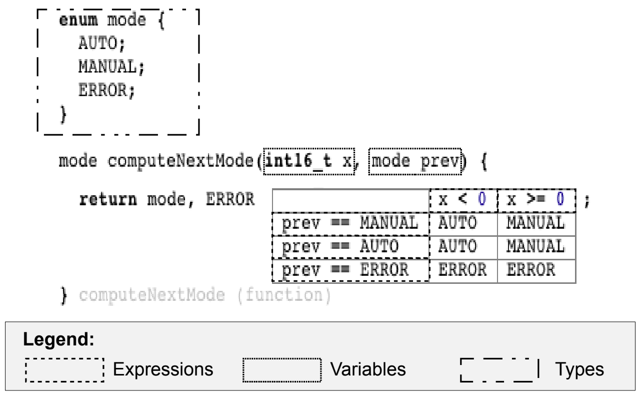
As Figure 4 shows, mbeddr can be seen as a matrix. On the horizontal-axis it is separated into an implementation concern (left side) and an analysis concern (right side). On the vertical axis it consists of a number of layers. At the center is MPS (the platform layer). On top of MPS, mbeddr ships with a number of core languages. On the implementation side, the core language is C. On the analysis side, mbeddr ships with languages that represent different analysis formalisms. Currently these comprise languages for specifying the input for SMT solvers and for specifying the input for model checkers. The next layer up consists of default extensions. On the implementation side mbeddr ships C extensions for interfaces and components, physical units, state machines plus various smaller C extensions. On the analysis side, the default extensions include support for model checking state machines and for consistency and completeness checking of decision tables. Below the common platform JetBrains MPS, mbeddr integrates existing tools that process the models: the *gcc* C compiler for the implementation side, as well as the NuSMV[1] model checker and Yices[2] and CVC[3] SMT solvers. On top of the default extensions, users can develop their own application level DSLs. These typically make use of the languages provided in the core and the default extensions either by directly extending C or its default extensions or by embedding parts of them into new application level DSLs. This process is significantly simplified by MPS support for language reuse and composition.

3. EXAMPLES OF APPLICATION-LEVEL AND ANALYSIS LEVEL DSLS

In this section we present examples of domain specific languages that address fundamentally different domains. The first one (presented in Section 3.2) is an extension of C with bi-dimensional conditions and is a part of the mbeddr default extensions. The second one (presented in Section 3.3) is a language that is motivated completely independent from C for embedded programming and that is focused on defining energy management rules for smart buildings. In Section 3.4 we present SMT analysis level languages. While the decisions table language is fundamentally domain agnostic, the second language has a high degree of intentionality and the addressed domain is very narrow. These two languages, however, share a common sublanguage that contains expressions, variables and types. Furthermore, the analyses that can be performed on their programs are similar: checking if all cases are covered by conditions (completeness) and if there are different cases that are overlapping (consistency). These analyses can be easily encoded as SMT problems.

3.1 Expressions Language Modules

A particular language module may be part of several other languages. The shared module used by our example DSLs is the *expressions module.* Expressions are relevant because they can be found in almost any non-trivial language (in business rules, state machine guards or controlling intelligent buildings). While the expressions are fundamentally similar regarding syntax, editor support, typing rules and transformation to the analysis tool syntax, they also *differ* depending on their reuse context in the following ways. Regarding their **abstract syntax**, different languages allow different kinds of expressions: expressions valid in one language can be forbidden in another one, or a certain language might require additional kinds of expressions. For example, languages that include functions may support function calls, simpler languages may not. Regarding the **concrete syntax**, in different languages, expressions might have different concrete syntaxes. For example some languages use && and others & for *logical and.* Also, languages might use a prefix or infix notation (& a b vs a & b). Regarding **context sensitive constraints**, some languages allow only expressions of a certain form. For example, SMT solvers can deal only with linear arithmetics and thereby, even though multiplication is allowed, expressions of form $x*y$ where both x and y are variables are not allowed ($x+y$ or $x*2$ would be valid).



```
enum mode {
  AUTO;
  MANUAL;
  ERROR;
}

mode computeNextMode(int16_t x, mode prev) {

  return mode, ERROR
} computeNextMode (function)
```

	x < 0	x >= 0
prev == MANUAL	AUTO	MANUAL
prev == AUTO	AUTO	MANUAL
prev == ERROR	ERROR	ERROR

Figure 5: An example decision table. We have highlighted the different aspects of the reused language module.

3.2 Decision Tables

Decision tables exploit JetBrains MPS' projectional editor in order to represent two-level nested *if* statements as a table.

[1]http://nusmv.fbk.eu/
[2]http://yices.csl.sri.com/
[3]http://www.cs.nyu.edu/acsys/cvc3/

Figure 5 shows an example. Decision tables [4] let users describe the different result values for different combinations of input conditions. The rationale for tabular expressions is to let developers define the conditions more easily and to allow reviewers to directly gain an overview of varied sets of input conditions. Decision tables are translated into C essentially as an *if/else if/else* for the column headers, and nested in each branch, an *if/else if/else* for the row headers. In Figure 5 we marked the different language modules that are used in the definition of decision tables.

Analyzing Decision Tables. For a two-dimensional decision table, there are two obvious possible analyses. The first one is *completeness* which requires that every behavior of the system is explicitly modeled and no case is omitted: this enforces explicitly listing all the possible combinations of the input conditions in the table. The second analysis is *consistency*, which checks whether there are overlapping input conditions, meaning that several cases are applicable for a single input value (non-determinism).

Encoding the Analyses as SMT Problems. Given a table with n rows (r_i) and m columns (c_j), we can check its *completeness* by checking the satisfiability of the following formula (if satisfiable, then the table is incomplete).

$$\neg \bigvee_{i,j=1}^{n,m} (r_i \wedge c_j)$$

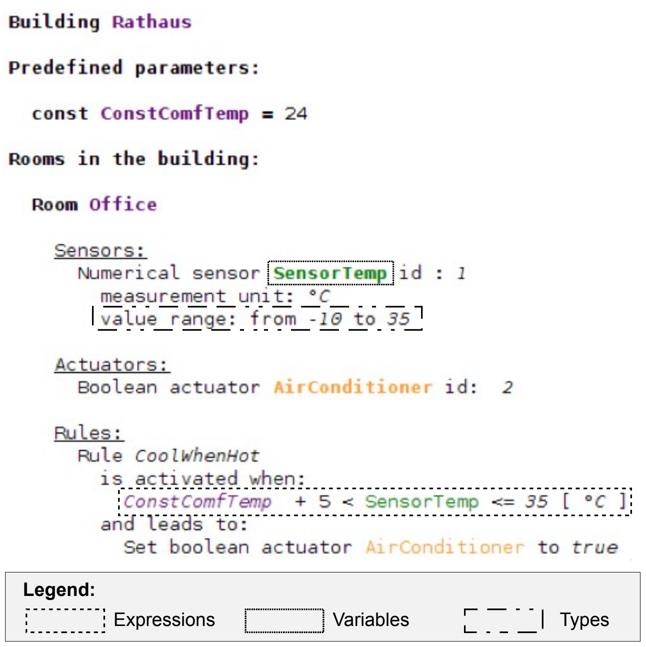
Similarily, the table is *inconsistent* if any of the formula is satisfied for $i, k \in \{1, ..., n\}$ or $j, l \in \{1, ..., m\}$ with $i \neq k \vee j \neq l$

$$(r_i \wedge c_j) \wedge (r_k \wedge c_l)$$

3.3 GRAPE

Intelligent buildings allow advanced optimizations like: improving energy consumption efficiency, maximizing the user comfort, creating special environmental conditions for a certain process (e. g., growing plants in a green house). Inteligent buildings let the administrator configure herself the concrete behavior by providing the system with a set of rules R, in accordance to which the building has to function. For example, if the temperature is above 27 °C and somebody is in the room, then switch on the air conditioner and set the desired air conditioner required temperature to be 24 °C.

Grape is a DSL built on top of MPS and that reuses parts of mbeddr C (see Section 4). It describes building, rooms, sensors and actuators and, finally, the desired set of rules R. A Cartesian product of all sensors ranges is called a sensor space S. Physical conditions, as measured by sensors, in a room at every discrete moment of time can be described as a point in the sensor space. Grape is integrated within an experimental demonstrator at ForTISS [5]. In Figure 6 we show how rules for smart buildings can be modeled with Grape. The analysis of the rules is needed, because the number of rules can be quite high (e.g. over 20 for just one room to control illumination) thus making it not trivial to maintain correctness of the rule system.



```
Building Rathaus

Predefined parameters:

  const ConstComfTemp = 24

Rooms in the building:

  Room Office

    Sensors:
      Numerical sensor SensorTemp id : 1
        measurement unit: °C
       value range: from -10 to 35

    Actuators:
      Boolean actuator AirConditioner id:  2

    Rules:
      Rule CoolWhenHot
        is activated when:
          ConstComfTemp  + 5 < SensorTemp <= 35 [ °C ]
        and leads to:
          Set boolean actuator AirConditioner to true
```

Legend:
Expressions   Variables   Types

Figure 6: Example of a Grape Model

Analyzing Grape Rules. For a set of rules described in Grape, we can define completeness and consistency analyses (similar to those for decision tables). A set of rules R is *complete* if and only if for each input from sensors in S the system "knows the required state" for each actuator A. A set of rules R is *consistent* if there is a clearly defined behavior of the building for each sensor state from S.

Encoding the Analyses as SMT Problems. Each rule $r_i \in R$ provides a mapping from a region $T \subseteq S$, where the condition (a Boolean formula) $c_i : S \rightarrow \{true, false\}$ of the rule is satisfied to some action a_i that will be performed by actuators. An actuator A can be set to values $v \in def(A)$. Each action causes a consequence, setting some of the actuators to defined values, e.g. set air conditioner (AC) power switch to on, set air conditioner required temperature to 24 °C. Every setting can be represented as a pair $(A, v) : v \in def(A)$. Let us denote a consequence for each action a_i as $cons(a_i)$ and define it in this way: an action a_i sets an actuator A to value v if and only if a pair $(A, v) \in cons(a_i)$. For the example rule above the consequence would be:

$$\{(AC\ switch,\ on), (AC\ required\ temp., 24°C)\}$$

For one actuator A completeness of a set of rules R is:

$$\forall s \in S \rightarrow \exists r_i \in R, v \in def(A) : (A, v) \in cons(a_i)$$

, which means that for every point in the sensor space there is a rule, setting the actuator, accordingly, to some value. Completeness of a set of rules R means that for *every* actuator A the set of rules R is complete.

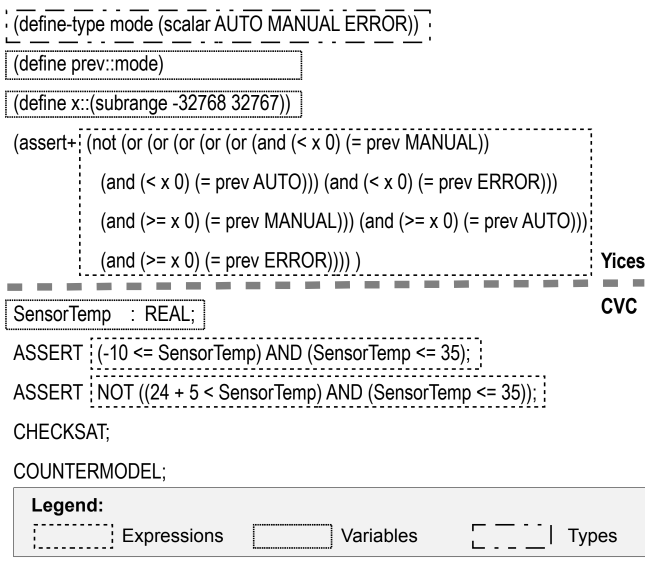
Consistency of two different rules $r_i, r_j \in R$ for an actuator A and $(A, vi) \in cons(r_i), (A, vj) \in cons(r_j)$ can be expressed, in turn, as

$$(c_i \wedge c_j \equiv false) \vee (vi \equiv vj)$$

This means that if two rules set the value for the same actuator, then they either set the actuator to the same value, or are never fired together in any single point of the sensor space. Consistency of a set of rules R means that for every actuator A the set of rules R is consistent. Collecting all the formulas for each actuator, negating them, and checking non-satisfiability is a translation of the Grape correctness problem into the SMT problem.

3.4 Yices and CVC

Yices and CVC3 are well known SMT solvers developed at SRI[4] and New York University[5]. The main motivation for integrating both of these tools is the fact that while Yices has better performance, it also has a proprietary and more restrictive license. As opposed to this, CVC3 is less performant but has a BSD license which allow us to ship it with the mbeddr stack which has an EPL license. In the upper-part of Figure 7 we show a fragment of a Yices file corresponding to the translation in SMT of the decision table from Figure 5. In the lower part of the figure is encoded in the CVC syntax the smart building from Figure 6. We can easy remark that the structure of these two files is the same and the main differences consist in the concrete syntax. In Figure 8 we show a fragment of a 'universal SMT language' (similar to SMT-LIB) which is tool independent.



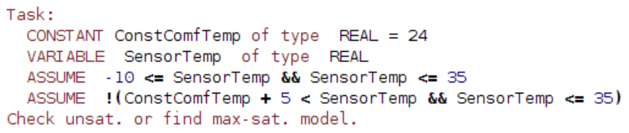
```
(define-type mode (scalar AUTO MANUAL ERROR))
(define prev::mode)
(define x::(subrange -32768 32767))
(assert+ (not (or (or (or (or (or (and (< x 0) (= prev MANUAL))
           (and (< x 0) (= prev AUTO))) (and (< x 0) (= prev ERROR)))
           (and (>= x 0) (= prev MANUAL))) (and (>= x 0) (= prev AUTO)))
           (and (>= x 0) (= prev ERROR)))) )
```

Yices
CVC

```
SensorTemp   : REAL;
ASSERT (-10 <= SensorTemp) AND (SensorTemp <= 35);
ASSERT NOT ((24 + 5 < SensorTemp) AND (SensorTemp <= 35));
CHECKSAT;
COUNTERMODEL;
```

Legend:
Expressions
Variables
Types

Figure 7: Example of Yices and CVC Files



```
Task:
  CONSTANT ConstComfTemp of type  REAL = 24
  VARIABLE  SensorTemp  of type  REAL
  ASSUME  -10 <= SensorTemp && SensorTemp <= 35
  ASSUME  !(ConstComfTemp + 5 < SensorTemp && SensorTemp <= 35)
Check unsat. or find max-sat. model.
```

Figure 8: Universal SMT: a language to capture commonalities between Yices and CVC

Observation. The Decision Tables, Grape and SMT languages make heavy use of expressions. Besides logical or arithmetics operators the expressions access constants and typed variables. As we showed above, the domain specific analyses are based on checking the satisfiability of different combinations of expressions which contain variables. Implementing the analyses implies simple transformations of the expressions from the application level DSLs into the Universal SMT language and then in the language of an SMT tool.

[4]http://yices.csl.sri.com
[5]http://www.cs.nyu.edu/acsys/cvc3/

4. IMPLEMENTATION

The mbeddr implementation of C itself is modularized into different language modules. One of them comprises primitive types, expressions and referenceable variables. The *Decision Tables*, *Grape* and *Universal SMT* language embed this expressions module defined as part of mbeddr C.

As a consequence of the projectional editor, very few concrete syntax concerns have to be considered when modularizing and composing languages. This means that that language development in MPS closely resembles object-oriented programming. Essentially all idioms and patterns known from mainstream OO languages like Java can be used in language development (details can be found in [9]). So in the remainder of this section we will just discuss the abstract syntax.

The expression language module defines an abstract concept *Type* and *Expression*. The primitive types are subconcepts of *Type* and are defined in the same expressions module. However, more specialized types (such as *NumericalSensor*) are defined in the *Grape* language (Figure 10). This is an example of language extension. A similar approach is used with subconcepts of *Expression* (Figure 9). For example, the decision tables language module defines the *DecisionTable* expression, which itself contains further expressions in the table cells.

Since the expressions language module has originally been defined for C, it contains a number of expressions that are specific to C, such as the bitwise operators. By reusing this language module in the universal SMT language, these expressions become available there as well, although they do not make sense in this context. MPS supports constraints that restrict the use of certain concepts in specific contexts: we have used one of these constraints to prevent users from using non-supported C expressions in the universal SMT language. This is an example of language restriction.

Another category of language elements shared between the application and analyses DSLs are variables. Each language implements its own kind of variables (e. g., local variables, global variables and parameters for Decision Tables, representing sensors for Grape, or constant declarations for

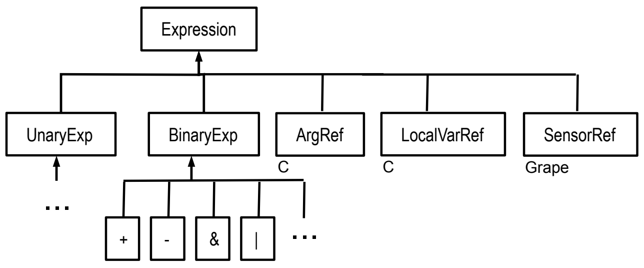


Figure 9: The hierarchy of expressions

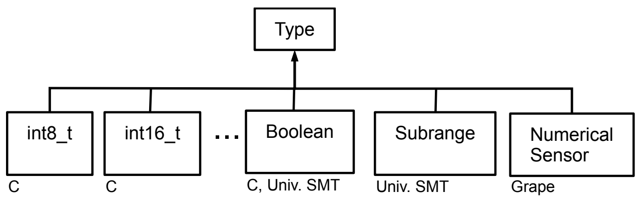


Figure 10: The hierarchy of types

Yices). The scoping mechanism for variables is implemented with the help of a *VariableScopeProvider* interface (Figure 11). This interface is responsible for providing the variables that are defined in a certain scope. Examples of scopes are *C_Function* or *C_File* in C programs, *Building* in Grape or *SMTFile* in the Universal SMT language.

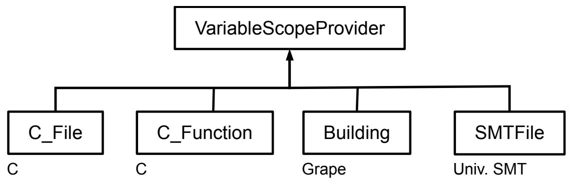


Figure 11: The hierarchy of scopes

Finally, there is a fine distinction between the decision tables and Grape languages. Decision tables are embedded in regular C programs, they feel like a C extension. The Grape language is a *separate language* that embeds C expressions. According to the classification in [9], both are examples of language Extension, because they have a dependency on the base language and they support syntactic mixing. However, Grape is an example of Extension with Embedding flavor: it feels like the C expressions are embedded into Grape.

5. RELATED WORK

Edwards and his colleagues [1] present an approach to automatically synthetize domain specific analysis tools. Their approach is to use properties attached to metaclasses in order to leverage additional semantics within metamodels. Thereby they enable generation of configuration files and plug-ins for extensible analysis and code generation frameworks. In this paper we described a complementary approach, namely how modularisation at the language level can be used to reuse different domain specific analyses in the context of different DSLs that use that language fragment.

[8] presents a methodology for creating DSLs focused towards verification. The authors capture domain specific knowledge and make explicit use of it in the verification. Our approach to reuse languages and analyses can be enriched with more domain knowledge. There is a lot of knowledge that belongs to the business domain of the language in which a sub-language is embedded and that can be used in making verification more efficient. For example, in the case of rules for managing smart buildings, the variables representing the temperature have in practice very narrow domains (typical temperature in a room is between -10 and 50 degrees centigrade with the extremes being highly improbable) and this limitation can highly speed-up analyses tools.

Merilinna and his colleagues [6] state that verification in the context of domain specific modeling comprises the following aspects: verification of the meta-model which captures the domain semantics, verification of the generators which maintain the semantic equivalence between input and output models and verification of textual generators. Our approach for building modular analyses based on modular sub-languages lowers the verification efforts since it can be done only once at a language module level even if that language module is further embedded in different DSLs.

6. CONCLUSIONS

Domain specific languages are languages dedicated to a certain domain. They allow domain experts to directly express their knowledge in a program at the abstraction level of the problem domain and without any implementation noise. Besides the improvements in usability and productivity, DSLs open new ways for defining, performing and using formal analyses. Implementing analyses for each DSL over and over again can be a tedious work. In this paper we advocate that many times analyses use only certain sub-languages that are shared among different DSLs and in such cases language and analyses can be reused between different DSLs. Thereby, the analyses implementation efforts for different DSLs can be avoided. We demonstrate this with the help of two DSLs that belong to different domains but that are amenable to the same kind of analyses based on SMT solving.

7. REFERENCES

[1] G. Edwards, Y. Brun, and N. Medvidovic. Automated analysis and code generation for domain-specific models. In *the joint 10th Working IEEE/IFIP Conference on Software Architecture and 6th European Conference on Software Architecture (WICSA/ECSA)*, 2012.

[2] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, 2012. to appear.

[3] E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley, 2004.

[4] R. Janicki, D. L. Parnas, and J. Zucker. *Tabular representations in relational documents*, pages 184–196. Springer-Verlag New York, Inc., 1997.

[5] D. Koss, F. Sellmayr, S. Bauereiss, D. Bytschkow, P. Gupta, and B. Schätz. Stablishing a smart grid node architecture and demonstrator in an office environment using the soa approach. In *First International ICSE Workshop on Software Engineering Challenges for the Smart Grid. I*, 2012.

[6] J. Merilinna and J. Pärssinen. Verification and validation in the context of domain-specific modelling. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, DSM '10, pages 9:1–9:6, New York, NY, USA, 2010. ACM.

[7] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[8] M. R. Phillip James. Designing domain specific languages for verification: First steps. In G. S. Peter Hofner, Annabelle McIver, editor, *ATE-2011 – Proceedings of the First Workshop on Automated*

Theory Engineering, volume 760 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[9] M. Voelter. Language and ide modularization, extension and composition with mps. In *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, 2011.

[10] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *Proceedings of SPLASH Wavefront 2012*, 2012.