FASTEN: An Open Extensible Framework to Experiment with Formal Specification Approaches

Conference Paper · May 2019

DOI: 10.1109/FormaliSE.2019.00013

Daniel Ratiu
Volkswagen AG

FASTEN: An Open Extensible Framework to Experiment with Formal Specification Approaches

Using Language Engineering to Develop a Multi-Paradigm Specification Environment for NuSMV

Daniel Ratiu
Siemens Corporate Technology
Munich, Germany
Email: daniel.ratiu@siemens.com

Marco Gario
Siemens Corporate Technology
Princeton, USA
Email: marco.gario@siemens.com

Hannes Schoenhaar
Siemens Corporate Technology
Munich, Germany
Email: hannes.schoenhaar@siemens.com

***Abstract*—Formal specification approaches have been successfully used to specify and verify complex systems. Verification engineers so far either directly use formal specification languages which can be consumed by verification tools (e.g. SMV, Promela) or main stream modeling languages which are then translated into formal languages and verified (e.g. SysML, AADL). The first approach is expressive and effective but difficult to use by non-experts. The second approach lowers the entry barrier for novices but users are limited to the constructs of the chosen modeling languages and thereby end up abusing the language to encode behaviors of interest.**

In this paper, we introduce a third approach that we call FASTEN, in which modular and extensible Domain Specific Languages (DSLs) are used to raise the abstraction level of specification languages towards the domain of interest. The approach aims to help novice users to use formal specification, enable experts to use multi-paradigm modeling, and provide tools for the developers of verification technologies to easily experiment with various types of specification approaches. To show the feasibility of the approach, we release an open-source tool based on Jetbrains' MPS language workbench that provides an extensible stack of more than ten DSLs, situated at different levels of abstraction, built on top of the SMV language. We use the NuSMV model checker to perform verification, to simulate the models and lift the traces at the abstraction level of the DSLs. We detail on the experience with designing and developing the DSLs stack and briefly report on using the DSLs in practice for the study of a communication protocol of a safety critical system.

***Keywords*-formal methods, domain specific languages**

I. Introduction

Formal methods have been successfully used for the specification and verification of complex systems. Formal tools can either be used directly or via modeling tools such as SysML/AADL. In order to directly use formal tools, engineers need to encode the concepts from their problem domain into the low level and verification oriented input language of the tool (Figure 1-left). For doing this, a big abstraction gap must be bridged and this requires specialized formal specification skills. Furthermore, the intent of high-level concepts gets lost along the encoding, and the resulting models are typically verbose and hard to review and change.

Another direction of work is to allow the engineers to work with their usual modeling languages and workflows (e.g. SysML, AADL) and then compile these models into lower level models (e.g. SMV, Promela) which can be consumed by verification tools (e.g. NuSMV, Spin) [1]–[4] (Figure 1-center). From the point of view of the verification engines, the modeling tools offer a convenient front-end for the engineers to express their models. This approach poses two challenges. Firstly, there are two encoding steps required – domain-level concepts must be modeled by engineers using the input language of the modeling tools and then a generic compiler translates the high level language constructs into the input language of the verification tool. Multiple encoding steps can make validation of the final translation very difficult, since the generated low code does not contain any reference to the original domain concepts. Secondly, usually only a part of the original modeling and target verification language are supported, leading to less explicit modeling and missed opportunities in terms both of clarity and performance.

On the right hand side of Figure 1, we present an intuitive overview of our proposed approach. We extend the input language of formal verification engines with modular Domain Specific Languages (DSLs) which allow engineers to raise the level of abstraction of the specification whenever a DSL construct fits better for the modeling task at hand. The stack of DSLs is open and can be extended with further DSLs to better capture and address particular needs of a business domain.

a) Goals: we are building FASTEN[1], an extensible formal specification environment, to pursue the following goals:

- Enable multi-paradigm, multi-notational and domain appropriate specifications which capture the intent of a problem domain at best in order to minimize the encoding gap and make specifications more explicit.
- Enable experimentation with the integration of different specification approaches which have been historically independently developed.
- Disseminate formal methods to industry practitioners by making formal specifications easier to write, understand, evolve and integrate in daily workflows.

FASTEN is an open source research tool which instantiates the approach for SMV, the input language of NuSMV model

[1]FASTEN stands for 'FormAl SpecificaTion ENvironment'

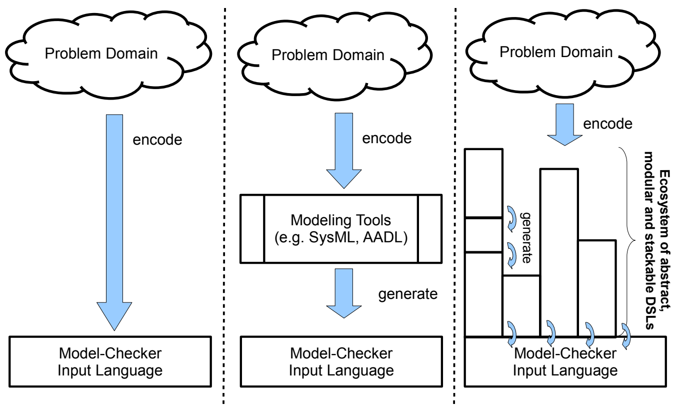

Problem Domain
encode
Model-Checker Input Language
Problem Domain
encode
Modeling Tools (e.g. SysML, AADL)
generate
Model-Checker Input Language
Problem Domain
encode
generate
generate
Ecosystem of abstract, modular and stackable DSLs
Model-Checker Input Language

Fig. 1. Using directly the formal verification tools requires bridging a big abstraction gap (left); using general purpose modeling tools involves two encoding steps (center); our proposed approach extends bottom-up the input language of a verification tool with higher level abstractions which are appropriate for a particular domain (right).

checker. To facilitate the experimentation in an industrial context, we took the decision to use exclusively free technologies. The binary distribution, including a tutorial, can be downloaded from the homepage[2], the sources from github[3].

b) Contributions: This paper brings the following contributions to the state of the art:

- an approach to build open and expressive formal specification environments around verification engines using stackable and modularly extensible domain specific languages.
- an extensible framework to experiment with the adequacy of different specification approaches on top of SMV, the input language of the NuSMV model checker.
- practical experience with instantiating the FASTEN approach, the feasibility of developing DSLs stacks and the potential benefits for the engineers using the DSLs.
- a showcase about synergies between language engineering and formal methods.

c) Structure: Section II presents the set of DSLs which are currently available in FASTEN. In Section III, we present at a glance the Jetbrains' MPS technology which is a key enabler for our approach. In Section IV, we present our preliminary experience with the proposed approach and, in Section V, we discuss variation points. The paper ends with the related work presented in Section VI and conclusions in Section VII.

II. FASTEN Language Stack

When modeling large systems, it is often the case that different parts of the system differ at the conceptual level. For example, a component might represent a state machine, another might encode a complex decision table, and yet another might simple exist to pack and unpack signals. When working in a setting where only state machines are allowed, we will have to encode the other two systems in a very unnatural way, making the model unnecessarily complex and hiding the original intent of the designer. FASTEN supports a *multiparadigm* approach by making it possible for all these modeling formalisms to co-exist.

In this section, we describe the initial set of DSLs provided by FASTEN on top of the SMV language. In each subsection we provide the motivation for the DSL, the key constructs, an example, and the translation into SMV. A key design decision in FASTEN is to keep the DSLs modular and this makes it possible to efficiently develop and experiment with new DSLs in an integrated environment.

A. SMV Base Language

The *SMV base language* is an implementation of the SMV language in the MPS language workbench. The base language is used as final step during the model-to-model transformations from more complex DSLs. Using the base language, FASTEN provides an IDE for writing SMV specifications and a front-end for the NuSMV tool.

B. Simple Extensions

Motivated by the feedback from our initial pilot, and from our previous experience with SMV, we provide several extensions on top of the base language which make it faster and easier to write SMV models. These extensions aim at avoiding repetitions, reducing inconsistencies in models, and leveraging IDE support such as auto-completion and on-the-fly consistency checks (e.g. type-checks).

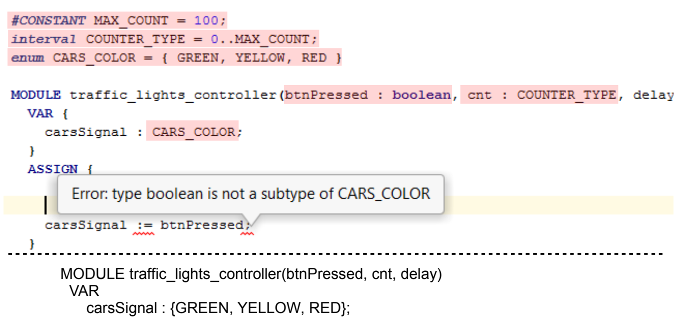
First, we enable *user defined types* and support explicit definition of enumeration, intervals and structured types. Furthermore, to make the purpose of types more explicit, we implemented a construct similar to `typedef` from C. Once the user defined types are defined, they can be used (and enforced) wherever types are required in SMV. Second, we enable first class support for defining *constants* and *simple functions* which are used in different parts of the model. Third, we extend the parameter declarations of SMV modules with *optional types*.

a) Example: In Figures 2- 4 we present examples of the simple extensions (marked in red). Figures 2 illustrates the definition and use of constants, definition and use of interval and enumeration types and the possibility to attach types to module parameters. Figure 3 illustrates the definition of aliases for different types, which in turn allows explicit modeling of the intent of a type. Type checks make use of this to prevent, for example, assigning variables representing lengths to those representing duration. Figure 4 illustrates the support for defining structured types. Richer types make the intent of models more explicit and they enable on the fly type-checks and thereby better tool support for the users to avoid errors.

b) Translation to SMV: In the lower pat of Figures 2- 4 we illustrate the translation to SMV. Types of parameters are deleted and the user defined types are inlined in the variables definitions. Typedefs are reduced to their core-types. Constants are folded and the value is inlined in the model. Structures are translated as modules, and members as variables of modules.

[2]https://sites.google.com/site/fastenroot/

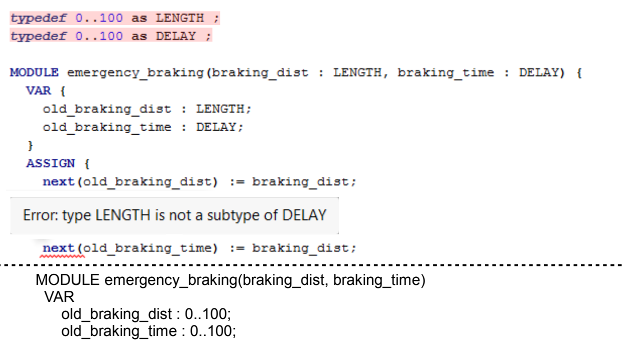
[3]https://github.com/mbeddr/mbeddr.formal



```
#CONSTANT MAX_COUNT = 100;
interval COUNTER_TYPE = 0..MAX_COUNT;
enum CARS_COLOR = { GREEN, YELLOW, RED }

MODULE traffic_lights_controller(btnPressed : boolean, cnt : COUNTER_TYPE, delay
  VAR {
    carsSignal : CARS_COLOR;
  }
  ASSIGN {
        Error: type boolean is not a subtype of CARS_COLOR
    |
    carsSignal := btnPressed;
  }
```

```
MODULE traffic_lights_controller(btnPressed, cnt, delay)
  VAR
    carsSignal : {GREEN, YELLOW, RED};
```

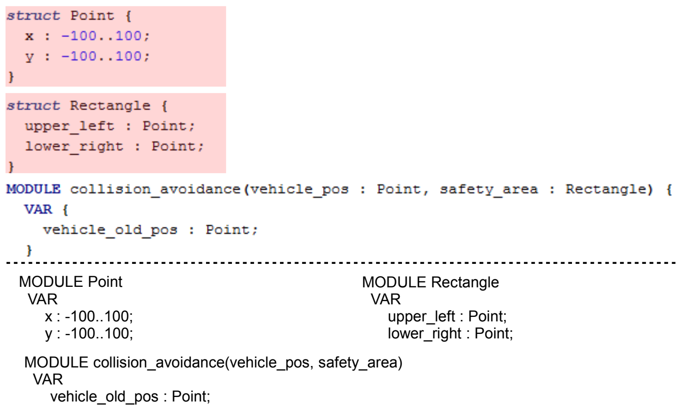
Fig. 2. Examples of user defined types, constants, and typed parameters which allow advanced type-checking.



```
typedef 0..100 as LENGTH ;
typedef 0..100 as DELAY ;

MODULE emergency_braking(braking_dist : LENGTH, braking_time : DELAY) {
  VAR {
    old_braking_dist : LENGTH;
    old_braking_time : DELAY;
  }
  ASSIGN {
    next(old_braking_dist) := braking_dist;
  Error: type LENGTH is not a subtype of DELAY
    next(old_braking_time) := braking_dist;
```

```
MODULE emergency_braking(braking_dist, braking_time)
  VAR
    old_braking_dist : 0..100;
    old_braking_time : 0..100;
```

Fig. 3. The `typedef` extension allows the definition of new types which are subsequently checked by the type-system.



```
struct Point {
  x : -100..100;
  y : -100..100;
}
struct Rectangle {
  upper_left : Point;
  lower_right : Point;
}
MODULE collision_avoidance(vehicle_pos : Point, safety_area : Rectangle) {
  VAR {
    vehicle_old_pos : Point;
  }
```

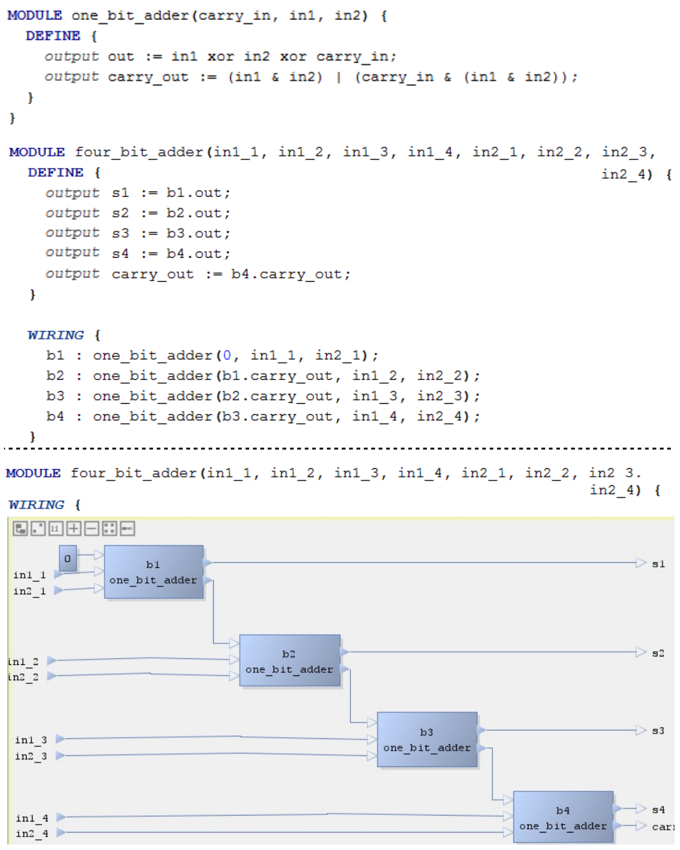
```
MODULE Point                          MODULE Rectangle
  VAR                                   VAR
    x : -100..100;                        upper_left : Point;
    y : -100..100;                        lower_right : Point;

MODULE collision_avoidance(vehicle_pos, safety_area)
  VAR
    vehicle_old_pos : Point;
```

Fig. 4. The `struct` extension allows the definition of structured types which make the intent of variables explicit.

C. Architecture

SMV does not explicitly support a component model. SMV provides modules which are similar to components, but the inputs and outputs are not explicitly defined. The formal parameters of a module can be used both as inputs and, due to the call-by-reference semantics of SMV, the parameters can be modified in a module and thereby used as outputs. Furthermore, a module can arbitrarily access the internal state or definitions of any submodule, thereby breaking encapsulation.

Encapsulation is desirable when creating larger models, and therefore we have extended the SMV language to allow explicit modeling of architectures. Components in the architecture are modules. The inputs of the components are the formal parameters of modules. The outputs are a special kind of `defines` which use the new `output` keyword. We



```
MODULE one_bit_adder(carry_in, in1, in2) {
  DEFINE {
    output out := in1 xor in2 xor carry_in;
    output carry_out := (in1 & in2) | (carry_in & (in1 & in2));
  }
}

MODULE four_bit_adder(in1_1, in1_2, in1_3, in1_4, in2_1, in2_2, in2_3,
                                                              in2_4) {
  DEFINE {
    output s1 := b1.out;
    output s2 := b2.out;
    output s3 := b3.out;
    output s4 := b4.out;
    output carry_out := b4.carry_out;
  }

  WIRING {
    b1 : one_bit_adder(0, in1_1, in2_1);
    b2 : one_bit_adder(b1.carry_out, in1_2, in2_2);
    b3 : one_bit_adder(b2.carry_out, in1_3, in2_3);
    b4 : one_bit_adder(b3.carry_out, in1_4, in2_4);
  }
```

```
MODULE four_bit_adder(in1_1, in1_2, in1_3, in1_4, in2_1, in2_2, in2 3.
                                                            in2_4) {
WIRING {
```

Fig. 5. The architecture DSL in textual notation (top) and diagramatic notation (bottom).

additionally enforce that components cannot write to any input (i.e., formal parameters of the module). A special type of component is a *composite component*, that is a restricted module which contains only two sections: i.e. `DEFINE` and `WIRING`. The `DEFINE` section contains the outputs definitions. In the `WIRING` section sub-components are declared. The actual parameters of the sub-components are either outputs of sibling modules or inputs of the parent module. We have defined two concrete syntaxes for composite components: a textual one (close to SMV), and a diagrammatic one which is close to a SysML blocks diagram and thus intuitive for practitioners. Depending on the use-case, it is possible to switch between the diagrammatic and textual syntax.

a) Example: In Figure 5 we present the architecture of a four-bit adder which is made up of four module instances which implement one bit adders. The one bit adder is defined through a `module` which has `outputs` as defines. The composite component contains a `WIRING` section which wires together the four one bit adders.

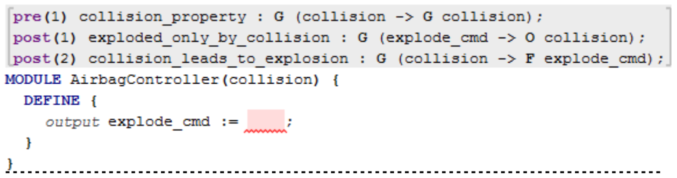
b) Translation to SMV: The outputs are translated to `DEFINE`s and the `WIRING` section to a `VAR` section, with component instances mapped to variables with module type.

D. Modules Contracts

SMV allows specification of properties inside a module (e.g. via `LTLSPEC`). There are cases when the expression inside a specification references only input parameters of the module.

In these cases specifications are effectively pre-conditions. Other specifications that reference only inputs and outputs but no internal variable are effectively post-conditions. Once we have defined syntactic interfaces of modules, an immediate extension is to allow the definition of contracts on modules.

a) Example: In Figure 6, we present an example of contracts for a module implementing an airbag controller. The module takes as input a Boolean value called `collision` representing the value of a sensor which registered a collision. The output is a command to provoke the airbag explosion. A precondition is that if the sensor registered a collision, the value of this input will keep its value. The post-conditions express the fact that the explosion is triggered if and only if a collision was sensed.



```
pre(1) collision_property : G (collision -> G collision);
post(1) exploded_only_by_collision : G (explode_cmd -> O collision);
post(2) collision_leads_to_explosion : G (collision -> F explode_cmd);
MODULE AirbagController(collision) {
  DEFINE {
    output explode_cmd :=      ;
  }
}
```

```
MODULE AirbagController(collision)                    LTLSPEC G (collision -> G collision);
  DEFINE                                              LTLSPEC G (explode_cmd -> H collision);
    explode_cmd :=  ...;                              LTLSPEC G (collision -> F explode_cmd);
```

Fig. 6. Contracts on modules can be explicitly defined in terms of pre-/post-conditions, and are subsequently translated into specifications at module level.

b) Translation to SMV: We translate the contracts to SMV as specifications written inside the corresponding module as illustrated in the lower part of Figure 6.

E. Contract-based Design

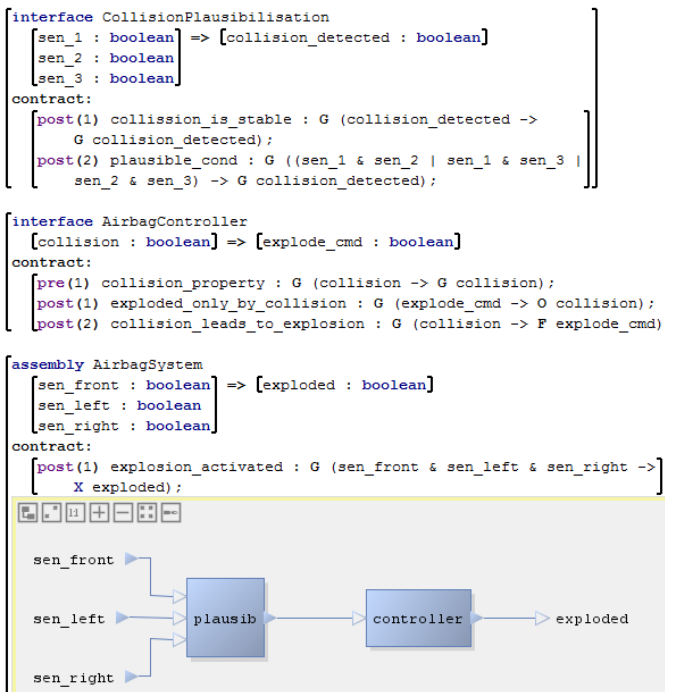
At the next abstraction level, we include the specification of architectures using contract-based design. This DSL is inspired by the powerful functionality of OCRA [5] and has the following constructs: `interfaces` specified with the help of typed input and output `ports`, contracts providing `pre-` and `postconditions` as LTL expressions, and `assemblies` which represent composition of components. `interfaces` can be put in relation to corresponding SMV `MODULE`s which are used to implement them.

a) Example: In Figure 7, we present an example of the architecture specified using contracts. The architecture contains two interfaces `CollisionPlausibilization` and `AirbagController`. These interfaces are assembled to implement the `AirbagSystem`.

b) Translation to SMV: We use the NuSMV engine to check the refinement of interfaces assembled into a component assembly. For doing this, we flatten each assembly and we synthesize an SMV `MODULE` which contains variables for all inputs and outputs, and a `DEFINE` enforcing the equations for the wiring of ports.

F. State Diagrams

The DSLs presented so far have focused on reducing modeling errors, and increasing the encapsulation of different modules. However, the implementation of the behaviors in the modules is mostly based on traditional SMV. The next two DSLs show two different paradigms for modeling behavior.



```
interface CollisionPlausibilisation
  sen_1 : boolean => collision_detected : boolean
  sen_2 : boolean
  sen_3 : boolean
contract:
  post(1) collission_is_stable : G (collision_detected ->
      G collision_detected);
  post(2) plausible_cond : G ((sen_1 & sen_2 | sen_1 & sen_3 |
      sen_2 & sen_3) -> G collision_detected);

interface AirbagController
  collision : boolean => explode_cmd : boolean
contract:
  pre(1) collision_property : G (collision -> G collision);
  post(1) exploded_only_by_collision : G (explode_cmd -> O collision);
  post(2) collision_leads_to_explosion : G (collision -> F explode_cmd)

assembly AirbagSystem
  sen_front : boolean => exploded : boolean
  sen_left : boolean
  sen_right : boolean
contract:
  post(1) explosion_activated : G (sen_front & sen_left & sen_right ->
      X exploded);
```

```
MODULE AirbagSystem___flattened(sen_front, sen_left, sen_right)
  VAR
    -- flattened architecture
    plausib_DOT_sen_1, plausib_DOT_sen_2, plausib_DOT_sen_3 : boolean;
    plausib_DOT_collision_detected, controller_DOT_collision : boolean;
    controller_DOT_explode_cmd : boolean;

  DEFINE
    arch_wiring := (sen_front = plausib_DOT_sen_1) & (sen_left = plausib_DOT_sen_2) &
                   plausib_DOT_collision_detected = controller_DOT_collision & ...;

  -- preconditions fulfillment by postconditions of previous components
  LTLSPEC (G  arch_wiring &
          (G (plausib_DOT_collision_detected -> G plausib_DOT_collision_detected) ->
          (G (controller_DOT_collision -> G controller_DOT_collision)
```

Fig. 7. An example of architecture and contracts. Contracts are translated to SMV code (lower part of the figure).

Symbolic transition systems are at the core of the SMV language. However, in many practical situations explicitly state machines is all we need to model a component. We support state machines with a DSL featuring two notations – the textual syntax, close to the SMV syntax, and the diagrammatic syntax, close to state charts and thereby more intuitive and easier to understand by practitioners. Our DSL requires an enumeration variable inside a module to be the *state* variable, with each enumeration value representing a possible state. The transitions of the state machine are implemented with a special kind of an `ASSIGN` section, named `STATEMACHINE`.

a) Example: In Figure 8, we present an example of a state machine which models the behavior of a simple traffic lights controller. The variable `traffic` is the state variable.

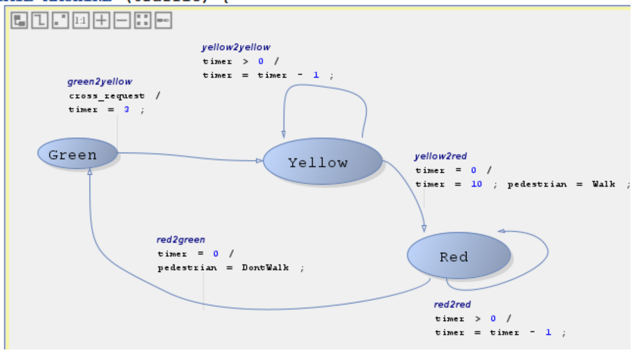
b) Translation to SMV: The translation to SMV follows the textual notation of the state machine. The guards of transitions defined in the `next` of the `state variable` are expanded in the corresponding `next` of the other variables.

G. Function Tables

Tabular specifications have been long proven to be highly expressive [6], and this motivated us to include them to specify the logic of modules. We defined a special construct called

```
MODULE traffic_lights_controller(cross_request) {
  VAR {
    state variable traffic : { Green, Yellow, Red };
    pedestrian : { Walk, DontWalk };
    timer : 0..10;
  }
 STATE-MACHINE (traffic) {
    init(traffic) := Green;
    next(traffic) := case
                       green2yellow: traffic = Green & cross_request : Yellow
                       yellow2yellow: traffic = Yellow & timer > 0 : Yellow;
                       yellow2red: traffic = Yellow & timer = 0 : Red;
                       red2red: traffic = Red & timer > 0 : Red;
                       red2green: traffic = Red & timer = 0 : Green;
                       otherwise: TRUE : traffic
                     esac;
    init(timer) := 0;
    next(timer) := case
                     guard_of(green2yellow) : 3;
                     guard_of(yellow2yellow) : timer - 1;
                     guard_of(yellow2red) : 10;
                     guard_of(red2red) : timer - 1;
                     otherwise: timer
                   esac;
    init(pedestrian) := DontWalk;
    next(pedestrian) := case
                          guard_of(yellow2red) : Walk;
                          guard_of(red2green) : DontWalk;
                          otherwise: pedestrian
                        esac;
  ,
```

```
MODULE traffic_lights_controller(cross_request) {
  VAR {
    state variable traffic : { Green, Yellow, Red };
    pedestrian : { Walk, DontWalk };
    timer : 0..10;
  }
  STATE-MACHINE (traffic) {
```



```
MODULE traffic_lights_controller(cross_request)
 VAR
  traffic : {Green, Yellow, Red};
  pedestrian : {Walk, DontWalk};
  timer : 0..10;
 ASSIGN
  init(traffic) := Green;
  next(traffic) := case
     traffic = Green & cross_request : Yellow;
     traffic = Yellow & timer > 0 : Yellow;
     traffic = Yellow & timer = 0 : Red;
     traffic = Red & timer > 0 : Red;
     traffic = Red & timer = 0 : Green;
     TRUE : traffic;
   esac;

  init(timer) := 0;
  next(timer) := case
     traffic = Green & cross_request : 3;
     traffic = Yellow & timer > 0 : timer - 1;
     traffic = Yellow & timer = 0 : 10;
     traffic = Red & timer > 0 : timer - 1;
     TRUE : timer;
   esac;
```

Fig. 8. A state machine in textual projection (top), graphical projection (center) and the translation to SMV (bottom).

`FUNCTION-TABLE` which is a specialization of the `DEFINE` section.

a) Example: In Figure 9, we present an example of a function table which implements a one bit full-adder. The `out` and `carry_out` are marked as outputs.

b) Translation to SMV: On the bottom of Figure 9 we present the translation of function tables via SMV `case` expressions. Each function definition is reduced into an `DEFINE`, enumerating all cases.

```
MODULE one_bit_adder(carry_in, in1, in2) {
  FUNCTION-TABLE {
```

				output out	output carry_out
carry_in = 0	in1 = 0	in2 = 0		0	0
		in2 = 1		1	0
	in1 = 1	in2 = 0		1	0
		in2 = 1		0	1
carry_in = 1	in1 = 0	in2 = 0		1	0
		in2 = 1		1	1
	in1 = 1	in2 = 0		0	0
		in2 = 1		1	1

```
MODULE one_bit_adder(carry_in, in1, in2)
 DEFINE
  out := case
    carry_in = 0 : case
      in1 = 0 : case
        in2 = 0 : 0;
        in2 = 1 : 1;
       esac;
     ...
```

Fig. 9. The logic of a module can be specified with function tables which are reduced to `DEFINE`s.

H. Unit Tests

DSLs presented so far have focused on improving the clarity of modeling. When creating components, it is important to show that each of them behaves as expected. The most common verification method among practitioners is testing. Our unit tests aim to enable users to quickly (partially) check and execute the specifications in an automated manner and to support best practices from software development like test driven development during the specification phase.

a) Example: Figure 10-left illustrates an example of a tabular unit test for a component implementing a counter which takes two inputs: a `stop_command` of boolean type which stops the counter when it is true and a `step` representing the current counting step. The outputs of this component are the validity of the output and the value of the current counter. Once the `stop_command` is true, then the outputs should not be valid any longer.

Figure 10-right illustrates a generalized unit test – '*' represents the cases when the inputs can take any value, and '#' represent any output values (don't care). Intuitively, the generalized unit test specifies that after `stop_command` takes

:est case: test example for module: counter

#	Inputs		Outputs	
	stop cmd	step	out valid	out value
1	FALSE	1	TRUE	0
2	FALSE	2	TRUE	1
3	TRUE	3	TRUE	3
4	FALSE	1	FALSE	6
5	FALSE	5	FALSE	6

test case: gen test example for module: c

#	Inputs		Outputs	
	stop cmd	step	out valid	out valu
1	FALSE	1	TRUE	0
2	FALSE	2	TRUE	1
3	TRUE	*	TRUE	3
4	*	*	FALSE	#
5	*	*	FALSE	#

```
MODULE main
 VAR
  __crtStep : 0..100;
  cnt : counter(stop_cmd,step);

 DEFINE
  stop_cmd := ((__crtStep = 0) ? FALSE:
                (__crtStep = 1) ? FALSE: ...;
  step := ((__crtStep = 0) ? 1: (__crtStep = 1) ? 2: ...
 ASSIGN
 init(__crtStep) := 0;
 next(__crtStep) := ((__crtStep + 1) < 5) ?
                    (__crtStep + 1):(5);

LTLSPEC ((cnt.output_valid=TRUE & cnt.output_value=0)) &
  (X (cnt.output_valid = TRUE & cnt.output_value = 1)) &
  (X X (cnt.output_valid = TRUE & cnt.output_value = 3)) ...
```

```
MODULE main
 VAR
  __crtStep : 0..100;
  cnt : counter(stop_cmd, step);
  stop_cmd_nondet : boolean;  step_nondet : 0..10;
 DEFINE
  stop_cmd := ... (__crtStep = 3) ? stop_cmd_nondet: ...
  step := ... ((__crtStep = 2) ? step_nondet: ...

 ASSIGN
 init(__crtStep) := 0;
 next(__crtStep) := (((__crtStep + 1) < 5) ?
                    (__crtStep + 1):(5));

LTLSPEC ...
 (X X (cnt.output_valid=TRUE & cnt.output_value=3)) &
 (X X X (cnt.output_valid = FALSE))...
```

Fig. 10. Unit tests (top-left) specify concrete inputs and expected outputs. Generalized unit tests (top-right) specifies don't care for both inputs (*) and outputs (#). The lower part presents the translation into SMV.

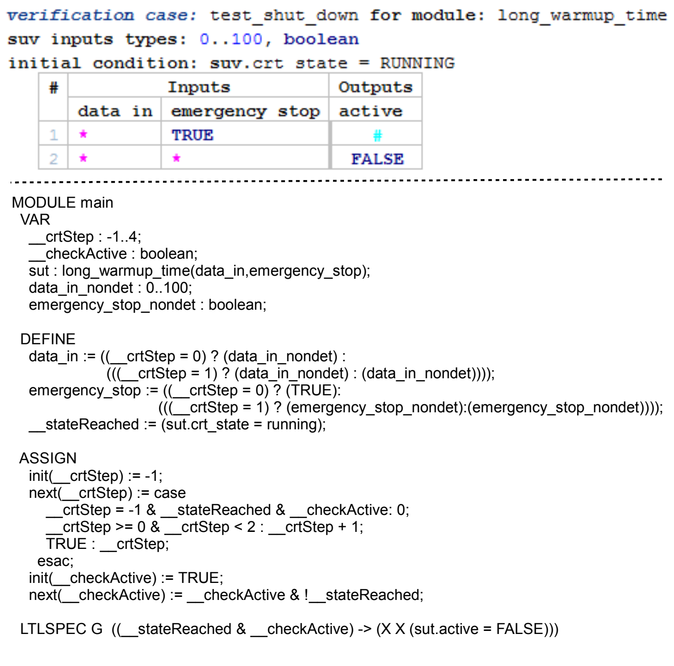
value TRUE, then the out_valid is FALSE independently of the values of step and stop_cmd.

b) Translation to SMV: For each unit test we generate a new .smv file which contains a copy of the module under test and a main module which acts as test driver. The test driver feeds the system under test with inputs which correspond to the unit-test. In the case of generalized test, for the inputs which are marked with '*' we generate an additional variable which takes non-deterministic values. The outputs which are marked with '#' are simply ignored by the oracle. Due to the fact that the unit test contains a fixed number of steps, we use the BMC engine of NuSMV for executing tests with a depth of the number of steps of the test-case.

I. Verification Cases

Many systems implement complex starting sequences consisting of booting, calibration and establishing communication with peers, therefore the engineers are interested in running a test case after the system has reached a certain state. To do this, we have designed a special construct called verification case which extends generalized unit tests to run after the system reached a certain state.

a) Example: The example system is initially in the WARMUP state for several steps, and then switches to the RUNNING state (as captured by the variable crt_state). After the system is in the RUNNING state, we would like to check that the system responds adequately to an emergency_stop command. For doing this, we define a verification case, illustrated in Figure 11-top which contains several inputs and expected outputs about the behavior of the system when it is running. In our example we expect that the system will be inactive, after the predicate suv.crt_state = RUNNING is satisfied, and subsequently the emergency_stop command is TRUE.



```
verification case: test_shut_down for module: long_warmup_time
suv inputs types: 0..100, boolean
initial condition: suv.crt state = RUNNING
```

#	Inputs		Outputs
	data in	emergency stop	active
1	*	TRUE	#
2	*	*	FALSE

```
MODULE main
  VAR
    __crtStep : -1..4;
    __checkActive : boolean;
    sut : long_warmup_time(data_in,emergency_stop);
    data_in_nondet : 0..100;
    emergency_stop_nondet : boolean;

  DEFINE
    data_in := ((__crtStep = 0) ? (data_in_nondet) :
                  (((__crtStep = 1) ? (data_in_nondet) : (data_in_nondet))));
    emergency_stop := ((__crtStep = 0) ? (TRUE):
                        (((__crtStep = 1) ? (emergency_stop_nondet):(emergency_stop_nondet))));
    __stateReached := (sut.crt_state = running);

  ASSIGN
    init(__crtStep) := -1;
    next(__crtStep) := case
        __crtStep = -1 & __stateReached & __checkActive: 0;
        __crtStep >= 0 & __crtStep < 2 : __crtStep + 1;
        TRUE : __crtStep;
      esac;
    init(__checkActive) := TRUE;
    next(__checkActive) := __checkActive & !__stateReached;

  LTLSPEC G  ((__stateReached & __checkActive) -> (X X (sut.active = FALSE)))
```

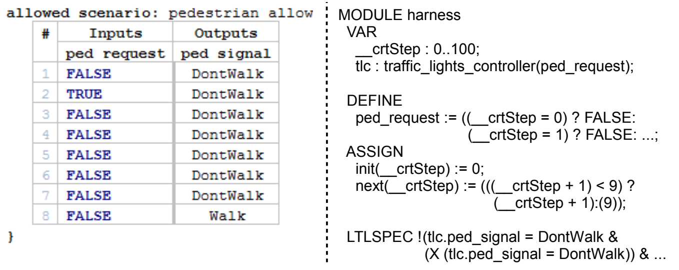
Fig. 11. Example of a verification case (top) and translation to SMV (bottom).

b) Translation to SMV: On the lower part of Figure 11, we sketch the translation to SMV of this verification case. The specification checks the actual and expected outputs of the after the system has reached the RUNNING state.

J. Scenarios (Un-)Feasibility

For the cases when the system under test exhibits non-determinism, unit tests are inadequate since there can be several outputs allowed for the same set of inputs. We implement the possibility to describe scenarios and check for their feasibility. For doing this, we introduce allowed scenario and disallowed scenario constructs which use tables containing the same entities like the unit tests (i.e. concrete values, "any value" for input or "don't care" for output).

a) Example: The requirements specification of an intelligent traffic lights controller for pedestrians can stipulate the possibility of the controller to respond to pedestrians' requests to cross the road after 4 and 10 steps depending on the traffic load. Figure 12 presents an example of an allowed scenario specification – after a pedestrian request is received, he is allowed to cross the road after 6 ticks.



```
allowed scenario: pedestrian allow
```

#	Inputs	Outputs
	ped request	ped signal
1	FALSE	DontWalk
2	TRUE	DontWalk
3	FALSE	DontWalk
4	FALSE	DontWalk
5	FALSE	DontWalk
6	FALSE	DontWalk
7	FALSE	DontWalk
8	FALSE	Walk

```
}
```

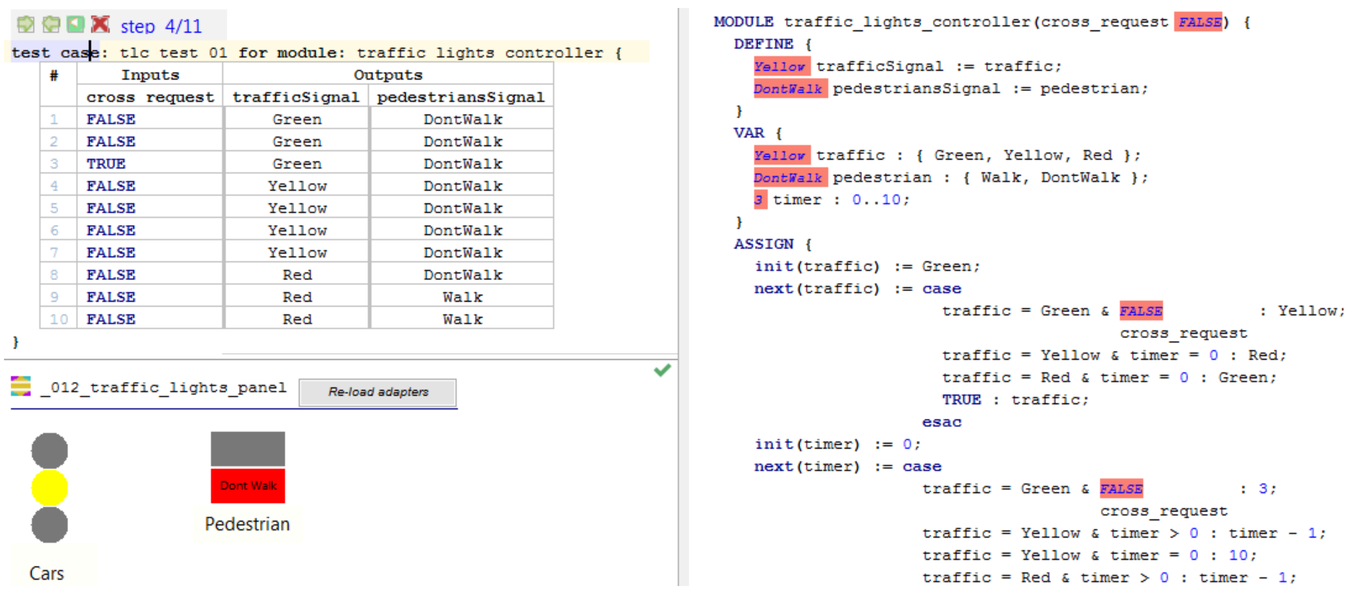
```
MODULE harness
  VAR
    __crtStep : 0..100;
    tlc : traffic_lights_controller(ped_request);

  DEFINE
    ped_request := ((__crtStep = 0) ? FALSE:
                      (__crtStep = 1) ? FALSE: ...;
  ASSIGN
    init(__crtStep) := 0;
    next(__crtStep) := (((__crtStep + 1) < 9) ?
                          (__crtStep + 1):(9));

  LTLSPEC !(tlc.ped_signal = DontWalk &
              (X (tlc.ped_signal = DontWalk)) & ...
```

Fig. 12. Allowed scenario (left) and its translation to SMV (right).

b) Translation to SMV: On the right side of the figure we present the SMV translation of this specification. If after running NuSMV, the property fails, this means that a trace has been found and thereby the scenario is feasible. If the property holds, then the scenario is not feasible and thereby an "allowed scenario" will fail. The disallowed scenarios have the same translation but they hold when the property holds.

K. Model-level Simulator

Finally, we included capabilities to better understand the dynamic behavior of the system as obtained, e.g., from a counterexample trace. To do so, we have developed a simulator which allows the users to perform step-wise executions of models, and animates them according to the state of the system. Part of the simulator is a DSL to describe graphical domain specific widgets which can be linked to variables of a model. During the simulation, the widgets are animated as well to reflect the state of the model at domain level. In Figure 13, we illustrate an example of how the simulator works when started on a set of inputs defined in a test-case. Users can step through the model and follow the change of the system state, both on the SMV model, and in the graphical widgets.



test case: tlc test 01 for module: traffic lights controller {

#	Inputs	Outputs	
	cross request	trafficSignal	pedestriansSignal
1	FALSE	Green	DontWalk
2	FALSE	Green	DontWalk
3	TRUE	Green	DontWalk
4	FALSE	Yellow	DontWalk
5	FALSE	Yellow	DontWalk
6	FALSE	Yellow	DontWalk
7	FALSE	Yellow	DontWalk
8	FALSE	Red	DontWalk
9	FALSE	Red	Walk
10	FALSE	Red	Walk

}

```
MODULE traffic_lights_controller(cross_request FALSE) {
  DEFINE {
    Yellow trafficSignal := traffic;
    DontWalk pedestriansSignal := pedestrian;
  }
  VAR {
    Yellow traffic : { Green, Yellow, Red };
    DontWalk pedestrian : { Walk, DontWalk };
    3 timer : 0..10;
  }
  ASSIGN {
    init(traffic) := Green;
    next(traffic) := case
                         traffic = Green & FALSE          : Yellow;
                                           cross_request
                         traffic = Yellow & timer = 0 : Red;
                         traffic = Red & timer = 0 : Green;
                         TRUE : traffic;
                       esac
    init(timer) := 0;
    next(timer) := case
                       traffic = Green & FALSE          : 3;
                                         cross_request
                       traffic = Yellow & timer > 0 : timer - 1;
                       traffic = Yellow & timer = 0 : 10;
                       traffic = Red & timer > 0 : timer - 1;
```

Fig. 13. Example of the simulator in action when run on unit tests. Current values of inputs and state variables are annotated in the model (red background on the right) and the outputs can be linked to a graphical widgets which display the state of the system and thereby make the validation easier (bottom-left).

III. Enabling Language Engineering

Our work relies on language engineering technologies, which refer to defining, extending and composing programming and domain-specific languages (DSLs) and tooling. Language workbenches are tools that support efficient language engineering. Our work is based on the language workbench MPS[4] which supports all aspects of the definition of DSLs such as abstract syntax, advanced editors, context-sensitive constraints, code generators, and analysers. To make the paper self contained we briefly enumerate below the core features of MPS which we use to build FASTEN, a detailed presentation of MPS is beyond the scope of this paper and can be found in [7]–[9].

a) Domain Appropriate Notations: At the core of MPS is a *projectional editor*. In a projectional editor, the user's editing actions lead directly to changes in the abstract syntax tree. No grammar or parser is involved. Projection rules render a concrete syntax from the abstract syntax tree. By decoupling the abstract syntax from the concrete syntax, it becomes easier to provide multiple notations for the same language constructs, including non-textual notations such as tables or diagrams.

In FASTEN, we use textual and diagrammatic notations for representing architectures and state-machines (Figure 5, Figure 8), and tabular notations for function-tables (Figure 9).

b) Modular and Extensible Language Eco-systems: The abstract syntax used to define an MPS language is defined via a meta-model. The meta-model describes and puts in relation various entities in the language, as typically done in regular object-oriented design. Since no parser is involved, grammar ambiguities cannot happen and existing languages can be extended with new constructs in a modular fashion: e.g. to create the typed formal parameters we only needed to subclass the formal parameter construct. Thereby, MPS makes it effective to define new languages or to extend existing languages with new constructs in a modular manner.

As of today, the stack of DSLs developed in FASTEN includes more than 10 languages. In Figure 14 we present a fragment of the meta-model of FASTEN using a simplified class-diagram-like notation. An SMV `MODULE` contains one or several sections like `DEFINE`, `VAR` or `ASSIGN`. A `VAR` section contains variables declarations, each variable having a type like `boolean` or `module`. In the grayed region of the figure is a small fragment of the DSLs – e.g. an architecture `WIRING` section is a kind of `VAR` and contains `MODULE_INSTANCE`s which are variable declarations with module type.

c) Rich Context Sensitive Constraints: Beside new constructs, a language extension in MPS also contains different constraints which are essential mechanisms to increase the usability of the IDE by guiding users towards the creation of valid models. MPS offers different mechanisms for defining context sensitive constraints – e.g. scoping rules, typing equations or arbitrary checks written in Java. Some constraints prevent the construction of invalid models up-front; others lead

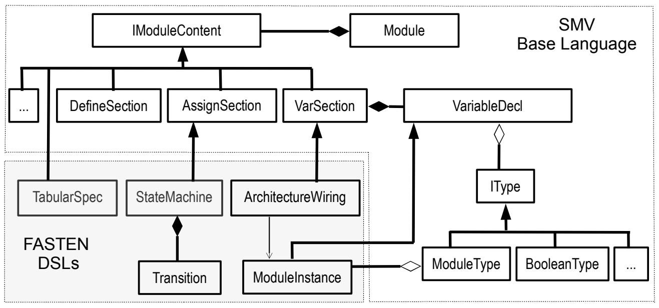


Fig. 14. A fragment of the meta-model of the top level entities of a module.

[4]https://www.jetbrains.com/mps/

to errors in the editor when they are violated.

FASTEN implements many constraints which effectively restrict SMV to fragments which represent idiomatic use of SMV – e.g. for defining composite components, the architecture DSL defines the `WIRING` section which essentially imposes constraints over the `VAR` section of SMV (detailed in Section II-C). Furthermore, using the rich constraints mechanisms of MPS, we have implemented the scoping rules of SMV and this helps the auto-completion functionality.

d) Modular and stackable model transformations: MPS offers two kinds of transformations: model-to-model transformations and text generation. Implementing our approach requires mostly a transformation of a model given in one language into a model of another language at a lower abstraction level. Only at the very end of the transformation chain a text generator is used to output a text representation used for downstream compilation. Because the transformations typically convert an abstract model to a more detailed model, DSLs and their model-to-model generators form a stack.

The SMV base language uses only text generators to produce ".smv" files from models. All other extensions on top of the SMV base language use model-to-model transformations to reduce their constructs to the SMV base language.

e) Advanced IDE and Tooling: MPS offers by default support for models authoring (auto-completion, syntax highlighting), for distributed development (diffing and merging at model level) and for an integration of external tools.

FASTEN features an advanced editor for SMV and all extensions, integrates deeply the NuSMV tool and lets the verification results be displayed directly in the IDE, counterexamples simulated and models animated. Furthermore, we support full traceability between high level models and the generated code to ease reviews and validation.

IV. Experience

We present our initial experience with the FASTEN approach by answering the following questions related to the feasibility of building the DSLs stack including associated tooling (Q1), and the end-user feedback about the use of the DSLs developed so far (Q2, Q3).

A. Building FASTEN

Q1) How feasible is it to extend the base language of a formal verification tool in a modular and open manner?: Implementing DSLs and associated tooling on top of a specification language of a verification engine requires effort. In the following, we provide an intuition on the efforts we invested for developing the current DSL stack.

We started with the implementation of FASTEN almost 2 years ago as a demonstrator project for the possible synergies between language engineering and formal methods. The entire DSL stack related to NuSMV contains the implementation of the most common concepts of the SMV language (covered roughly 75%) and 10 additional modular DSLs each organized in an MPS language module. The efforts for developing the languages and associated tooling (editors, generators, simulator) was ca. 6-9 person months. Most of the effort went into creating the infrastructure (i.e., implementation of the SMV base language, the integration of the NuSMV tool, importing existing SMV files). The integration of a new DSL requires skills both about language engineering with MPS and about the verification tool itself, and can take between several days and several weeks depending on scope and complexity.

Developing new DSLs with MPS can be done in a modular manner without having to modify other DSLs. Experimentation with different DSLs and their continuous refactoring and evolution based on the know-how gained from their usage are possible. DSLs can be evolved based on the evidence gathered from their use in the spirit of [10] and new DSLs can be developed to ease the specification work.

We conclude that by using the language engineering capabilities provided by MPS, it is feasible to develop modular and extensible DSLs on top of the input languages of verification engines and thereby to operationalize the FASTEN approach.

B. Using FASTEN

Q2) How are the DSLs received by practitioners?: One of the main motivations for our work is to disseminate formal methods to practitioners. A first step in the dissemination is the ability of novice users to understand the purpose of different DSL concepts and learn how to use them.

We have given extensive demos on FASTEN to practitioners from within different Siemens business units. They were able to understand the main concepts and relate to their previous modeling experience easily. Users immediately appreciate the diagrammatic notation of architectures, semantically rich interfaces via contracts, the possibility to model state-machines, function tables, and (generalized) unit tests. Being able to create tests, execute them, and animate the model drastically ease the understanding process.

Our experience so far is that higher level abstractions and notations familiar to the engineers are well received and appreciated. By starting with higher abstractions like state machines and function tables, it is possible to gradually introduce the users to more specific concepts of the underlying language, thus reducing the learning curve.

Q3) How do practitioners work with FASTEN?: We have piloted FASTEN in a realistic project about the specification of an end-to-end protection component for a communication protocol in the railway domain. The modeling was initially performed using components and state charts in another modeling tool. After that, the model was re-written in FASTEN using its various abstractions by a safety engineer without previous background about formal methods. The goal was twofold – firstly, investigate what abstractions increase the productivity for a certain modeling task by allowing our user to express his models "naturally" and, secondly, to perform verification on the model. The experiment took ca. two weeks spanned over two months.

The first step in the re-engineering was to re-create the architecture of the system using the FASTEN extensions.

Having the possibility to switch to diagrammatic notation for the architecture was very important for performing the first modeling. In the previous tool, there have been cases in which the logic of a component was modeled as a state machine with a single state and several transitions to this state. This type of degenerate state machine was an indication that the modeling formalism was not sufficient to capture the engineer intent. The same functionalities were modeled more explicit in FASTEN by using other abstractions like e.g. `FUNCTION-TABLE`. Learning about explicit modeling was a continuous process.

From the very beginning the pilot user started using the "simple" unit-testing capabilities of FASTEN. After several unit-tests it was clear that they had some redundancy in the sense that some inputs kept repeating while others were varying. This motivated us to develop the `generalized unit tests` by giving the possibility to specify that certain inputs can take "any value" from a range. Another redundancy was the fact that some unit tests were using the same set of steps as prefix in order to bring the system in a certain state – and only after this, different test steps were developed. This observation motivated us to extend the DSLs stack with the `verification case` construct which allows running a unit test after the system reached a certain state. The feedback loop of tuning the language to reflect the concerns of the pilot project took ca. two weeks of effort.

Our pilot user appreciated the editor automation and IDE functionality provided by FASTEN – this makes it possible to perform changes on models (e.g. renaming of entities, restructuring) and keeping them consistent. The integrated animation of counterexamples and traces of test-cases eased the validation of the specification. Very often practitioners simply "play" with the models by mutating and simulating them in order to increase their confidence that the specification reflects their intuition.

Having the possibility to keep users in the loop and experiment with new DSLs and extend the tooling to satisfy their modeling needs and reflect the intent in the models seems to us promising to increase the acceptance of formal models by practitioners. FASTEN is a framework which allows the experimentation with different specification approaches and is thereby an example of potential synergies between language engineering and formal methods.

V. Discussion

a) On the abstraction power of the FASTEN approach: We use language engineering technologies to build a stack of DSLs which increase the abstraction level of formal specification languages like SMV. Our DSLs reflect typical idiomatic usage of SMV for modeling and specification implemented using constructs which can be reduced to SMV via generators. One might ask himself how far can DSLs go when compared to SMV – e.g. could we implement full state-charts using this approach? Even if it is in principle possible, implementing abstractions which are "far away" from the base language is increasingly difficult and error prone. The decision about building more abstract DSLs is an engineering one which must consider the cost-benefits ratio.

b) On the experimentation with language abstractions: Most of the DSLs were developed in an iterative manner by considering the lessons learnt from their use. For this, the powerful functionality of MPS for language development has proved to be highly useful. Being able to quickly prototype DSLs and discuss them with our users and based on the feedback to evolve them proved to be essential. Furthermore, language modularization and extension capabilities of MPS are crucial enablers of our approach.

c) On extension possibilities: As of today, the FASTEN approach is instantiated for the SMV language and the NuSMV tool. Possible extensions are both the integration of additional formalisms, extending the stack of DSLs with new languages, or refining the available DSLs. Our tooling is available on github and new languages can be created by anyone – we are already aware of other research done outside of Siemens which uses FASTEN as basis for their approach.

d) On semantic integrity of languages and tooling: SMV has a formally defined semantics. The semantics of our DSLs is given by the underlying translation to SMV, and we rely strictly on NuSMV capabilities for both running unit tests, and performing verification and simulation of the models. This choice avoids complex semantical mappings and expensive theoretical and technical validation activities. Nevertheless, inconsistencies are still possible due to misunderstanding by the user of the semantics of the DSL constructs, and bugs in the tooling. We mitigate these situations by providing rich support for simulation and testing and, more importantly, by enabling a fine-grained traceability between the models and generated SMV text files. At any point in the model, users can see the generated SMV and thereby eventual generator bugs can be discovered. This strategy is most effective when the abstraction gap between the input and output languages is limited and traceability is clear.

VI. Related Work

a) DSLs and Formal Methods: A line of research uses formal methods to verify models described with domain specific languages [11], [12]. These approaches propose creation of DSLs for a specific business domain and take advantage of the domain concepts for easing the verification. Our approach uses modern language engineering to provide incrementally abstract domain specific constructs but still enabling the modeling of a broad category of systems.

b) IDEs for formal specification tools: There have been different efforts to provide modern IDEs support for formal specification tools [13]–[15]. These works provide a front-end to the formal specification tool by offering modern IDE support like auto-completion or syntax highlighting for writing the models. [16] presents a DSLs based approach to write extensible specifications of Promela by lifting common idioms of Promela at language level. We advance on this by using modular and extensible DSLs to increase the abstraction level at which specifications are performed. Our goal is to build a framework which enables the integration of different notations, multi-paradigm modelling and experimentation with the usability of different well-established specification approaches.

c) Specification Comprehension: There is a line of research which deal with the difficult challenge of comprehensibility of formal specifications [17], [18] in the context of systems maintenance and evolution. One of their main findings is that making dependencies between different specification parts explicit and using guidelines to write specifications can help to increase their comprehensibility. Our approach captures idioms of using formal specification languages and makes the intent more explicit with the help of higher-level DSLs. Furthermore, high-level constructs like the architecture, contracts or state-diagrams make the dependencies between specification fragments explicit. From this point of view, our work can be seen as tooling which implements the lessons learnt from empirical studies in order to facilitate the comprehension of specifications. This is somehow in contrast with the many works that start from a high-level language and translate it to SMV [19], since the focus is ultimately on usability as in [20], [21].

d) DSLs built with MPS: We have already used MPS for building MBEDDR, a fully featured IDE for C development and integrate formal verification approaches in it [22]–[24]. Despite the higher level of abstractions in MBEDDR, it remains a software development tool which would cannot fit the requirements of system-level modeling. In this paper we focus on DSLs stacks which increase the abstraction level and enables multi-paradigm modeling for the NuSMV model checking tool. Besides developing the stack of languages over SMV, FASTEN is however a more generic approach which can be instantiated also for other tools.

VII. Conclusions and Future Work

FASTEN is part of a longer endeavor at Siemens Corporate Technology to develop tooling, methods and approaches which enable practicing engineers to benefit from the power of formal methods. In this paper we introduced FASTEN as an open, modular and extensible framework to experiment with formal specification approaches and their potential for integration. FASTEN represents a new philosophy of building integrated specification environments with the help of stacks of DSLs built on top of the input language of verification tools. We demonstrate the feasibility of the approach with the help of ten DSLs which we developed on top of SMV and by integrating the NuSMV model checker.

Our future work goes along three directions. Firstly, we plan to experiment with how engineers use different specification notations and approaches in different business domains and for different use-cases; secondly, we plan to extend the DSL stack with further DSLs which capture specification patterns specific for a class of systems (e.g. fault tolerant) and from a certain business domain (e.g. rail, medical, automotive); and, thirdly, we plan to develop and extend similar stacks on top of the input language of other verification tools like NuXmv, Spin, Prism or Z3.

References

[1] G. Caltais, F. Leitner-Fischer, S. Leue, and J. Weiser, "Sysml to nusmv model transformation via object-orientation," in *Cyber Physical Systems. Design, Modeling, and Evaluation*. Springer, 2017.

[2] S. Kanav and V. Aravantinos, "Modular transformation from af3 to nuxmv," in *International Workshop on Model-driven Engineering Verification and Validation (MoDeVVa)*, 2017.

[3] A. Abdulhameed, A. Hammad, H. Mountassir, and B. Tatibouet, "An approach to verify sysml functional requirements using promela/spin," in *International Symposium on Programming and Systems (ISPS)*, 2015.

[4] T. Ando, Y. Miyamoto, H. Yatsu, K. Hisazumi, W. Kong, A. Fukuda, Y. Michiura, K. Sakemi, and M. Matsumoto, "Sysml state machine diagram to simple promela verification model translation method," in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, 2016.

[5] A. Cimatti, M. Dorigatti, and S. Tonetta, "Ocra: A tool for checking the refinement of temporal contracts," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013.

[6] A. Wassyng and R. Janicki, "Tabular expressions in software engineering," in *International Conference on Software and Systems and their Application (ICSSEA)*, 2003.

[7] F. Campagne, *The MPS Language Workbench*. CreateSpace Publishing, 2014.

[8] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering*. dslbook.org, 2013.

[9] M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen, "Lessons learned from developing mbeddr: a case study in language engineering with mps," *Software & Systems Modeling*, 2017.

[10] J. van den Bos and T. van der Storm, "A case study in evidence-based dsl evolution," in *Modelling Foundations and Applications*, 2013.

[11] J. Bodeveix, M. Filali, J. L. Lawall, and G. Muller, "Formal methods meet domain specific languages," in *5th International Conference on Integrated Formal Methods, IFM*, 2005.

[12] P. James and M. Roggenbach, "Designing domain specific languages for verification: First steps," in *Proceedings of the First Workshop on Automated Theory Engineering ATE*, 2011.

[13] P. Arcaini, A. Gargantini, and E. Riccobene, "Nuseen: A tool framework for the nusmv model checker," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017.

[14] B. de Vos, L. C. L. Kats, and C. Pronk, "EpiSpin: An Eclipse plug-in for Promela/Spin using Spoofax," in *International Workshop on Model Checking Software (SPIN)*, 2011.

[15] Z. Brezocnik, B. Vlaovic, and A. Vreze, "SpinRCP: the Eclipse rich client platform integrated development environment for the Spin model checker," in *International Symposium on Model Checking of Software (SPIN)*, 2014.

[16] Y. Mali and E. V. Wyk, "Building extensible specifications and implementations of Promela with AbleP," in *Workshop on Model Checking Software (SPIN)*, 2011.

[17] A. Bollin and D. Rauner-Reithmayer, "Formal specification comprehension: The art of reading and writing z," in *Proceedings of the Workshop on Formal Methods in Software Engineering*, 2014.

[18] A. Bollin, "Concept location in formal specifications," *Journal od Software Maintenance and Evolution*, 2008.

[19] E. Clarke and W. Heinle, "Modular translation of statecharts to smv," Tech. Rep., 2000.

[20] D. Méry, "Modelling by patterns for correct-by-construction process," in *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*. Springer, 2018.

[21] T. C. Ruys, "Low-fat recipes for SPIN," in *International Workshop on SPIN Model Checking and Software Verification*. Springer, 2000.

[22] D. Ratiu, B. Schaetz, M. Voelter, and B. Kolb, "Language engineering as an enabler for incrementally defined formal analyses," in *1st International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*, 2012.

[23] M. Voelter, D. Ratiu, B. Kolb, and B. Schätz, "mbeddr: instantiating a language workbench in the embedded software domain," *Automated Software Engineering*, 2013.

[24] D. Ratiu and A. Ulrich, "Increasing usability of Spin-based C code verification using a harness definition language," in *International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017, 2017.