ProMoBox in Practice : A Case Study on the GISMO Domain-Specific Modelling Language

Romuald Deshayes[1], Bart Meyers[2], Tom Mens[1], and Hans Vangheluwe[2,3]

[1] Département d'Informatique, Université de Mons, Mons, Belgium
firstname.lastname@umons.ac.be
[2] Modeling, Simulation and Design Lab (MSDL), University of Antwerp, Belgium
firstname.lastname@uantwerp.be
[3] Modeling, Simulation and Design Lab (MSDL), McGill University, Canada

Abstract. Domain-specific modelling (DSM) helps designing systems at a higher level of abstraction, by providing languages that are closer to the problem space than to the solution space. Unfortunately, specifying and verifying properties of the modelled system has been mostly neglected by DSM approaches. At best, this is only partially supported by translating models to formal representations on which properties are specified and evaluated based on logic-based formalisms. This contradicts the DSM philosophy as domain experts are usually not familiar with such formalisms. To overcome this shortcoming, the *ProMoBox* approach lifts property specification and verification tasks up to the domain-specific level. For a given DSM language, some operations at the metamodel level are needed to allow specification and verification of properties. This paper reports on a practical case study of how to apply the *ProMoBox* approach on *GISMO*, a DSM language designed specifically for developing gestural interaction applications.

1 Introduction

Domain-specific modelling (DSM) helps designing systems at a higher level of abstraction. By providing languages that are closer to the problem domain than to the solution domain, low-level technical details can be hidden. An essential activity in DSM is the specification and verification of properties to increase the quality of the designed systems [1]. Providing support for these tasks is therefore necessary to provide a holistic DSM experience to domain engineers. Unfortunately, this has been mostly neglected by DSM approaches. At best, support is limited to translating models to formal representations on which properties are specified and evaluated with logic-based formalisms [2], such as Linear Temporal Logic (LTL). This contradicts the DSM philosophy as domain experts desiring to specify and verify domain-specific properties are not familiar with such formalisms.

In previous work we proposed the *ProMoBox* framework to shift property specification and verification tasks up to the DSM level. The scope, assumptions and limitations of this approach are presented in [3]. The *ProMoBox* framework consists of (i) generic languages for modelling all artefacts that are needed for specifying and verifying properties, (ii) a fully automated method to specialise and integrate these generic languages in a given DSML, and (iii) a verification backbone based on model checking that is directly pluggable to DSM environments such as AToMPM [4]. Properties in *ProMoBox* are translated to LTL and a Promela model is generated that includes a translation of the system, its environment and its rule-based operational semantics. The Promela model is checked with the SPIN model checker [5] and if a counter-example is found it is translated back to the DSM level.

This paper presents a case study that applies *ProMoBox* to *GISMO*, a DSML for executable modelling of gestural interaction applications. We illustrate how to make some minor changes and additions to the metamodel of the DSML in order to enable the generation of all needed languages (an input language, output language, property language, and runtime language) that are required for the specification and verification of properties at DSM level. We subsequently report on the results of verifying these properties.

The paper is structured as follows. Section 2 presents the *GISMO* DSML used as a case study. Section 3 explains the changes required to the *GISMO* metamodel in order to apply the *ProMoBox* approach. Section 4 presents the results of applying *ProMoBox* to *GISMO*. Section 5 exemplifies the specification of domain-specific properties on *GISMO* models and provides some results and counter-examples found after verification of these properties. Section 6 presents related work, and Section 7 concludes.

2 *GISMO*: a DSML for gestural interaction

GISMO is a DSML developed by the first author to facilitate development of gesture-based interactive applications [6]. Specifying gestural interaction with objects is achieved in a state-based way. The *GISMO* metamodel is depicted in Fig. 1. A *GISMO* model is composed of states and gestures. State changes may be performed when a gesture is performed by a user. In previous work we have developed a framework [7] that takes care of interpreting such high level gestures from the raw data coming from 3D sensors such as Microsoft's Kinect motion sensor. The operational execution semantics of *GISMO* is based on ICO [8], a formalism based on high-level Petri nets.

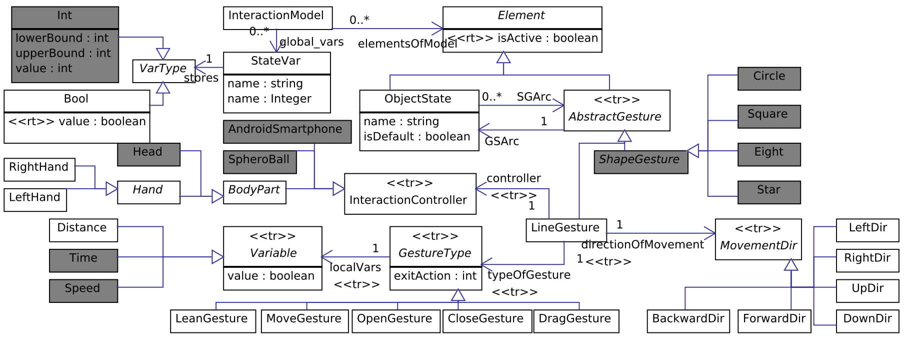


Fig. 1. Metamodel of the *GISMO* DSML for gestural interaction.

Fig. 2 shows an example of a domain-specific model in *GISMO*, representing the gestural interaction of a user with a virtual bow as part of some computer game. States are represented as labelled rounded rectangles. The unique active state is displayed in green. Gesture boxes are used to specify the gesture that is expected from the user. They are composed of the body part (e.g., left or right hand) involved in the gesture, the direction of movement (e.g., left, right, up, down), the type of gesture (e.g., moving, dragging, opening) and the observed dimension of the gesture (i.e., distance d, time t or speed s).

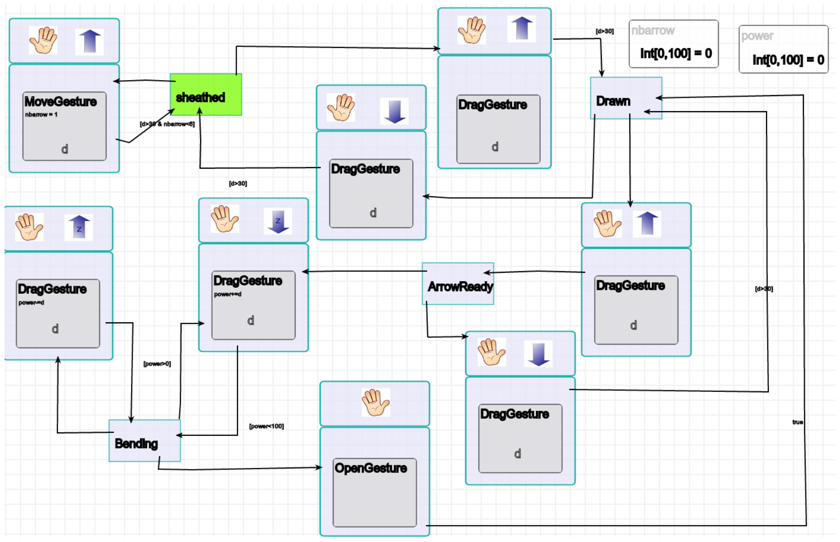


Fig. 2. GISMO model representing the expected gestural interaction of a user with a virtual bow.

The model is executed as follows. Whenever the user performs a specific gesture matching an expected outgoing gesture from the currently active state, the active state changes to the destination of the outgoing edge of the performed gesture. Global variables can be defined on *GISMO* models (e.g. nbarrow and power in Fig. 2). They store relevant information about the object being modeled. Triggering a state change can be conditioned by a boolean precondition that may combine the observed dimension of the gesture and the global variables. For example, a state change from *Drawn* to *ArrowReady* is triggered if the distance d of the *drag* gesture exceeds 30cm and the global variable nbarrow is strictly positive. Exit actions of a matching gesture can be executed to perform operations on a global variable, such as reassigning its value by the value retrieved from the observed dimension. In the example of Fig. 2, the two *drag* gestures connected to state *Bending* add or subtract from global variable power the value of the gesture's distance dimension.

3 Simplifying the *GISMO* metamodel

Enabling verification of domain-specific model properties requires a certain amount of preparatory work in order to make it applicable in practice. In particular, the *ProMoBox* approach [3] requires the metamodel of the DSML under study to be annotated before starting the generation process. Depending on the complexity of the language, other changes may be required as well. Most of the simplifications aim to reduce the combinatorial search space of the SPIN model checker. Not performing such simplifications would result in an exponentially longer verification time. This scalability problem is inherent to model checking, as the search space grows very fast in terms of the number of different possible inputs and input types.

This section focuses on the changes and simplifications required to the *GISMO* metamodel in order to enable specification and verification of domain-specific properties. A first simplification ignores the *time* and *speed* dimensions of input gestures. Thus, we restrict ourselves to properties based on the *distance* dimension only. Moreover, as unbounded variable domains are too costly to check, we provided a binary classification of a gesture's *distance* value into either small or big. A wider range of distance classes could be used, at the expense of a longer verification time.

As another simplification we limit the types of global variables to boolean values only, i.e., integer values are not supported during property verification. We also represent each global variable by a unique integer identifier instead of a variable name since strings are not supported by Promela, the verification modelling language to which our DSML models are translated. This is not a huge concern since it can be fully automated and made transparent for the DSML engineer.

Finally, we simplified the preconditions and exit actions of *GISMO* models. In *GISMO*, preconditions are boolean expressions resulting from the logical composition (*AND, OR, NOT*) of comparisons between global variables, gesture dimensions and integer values (*e.g.,* nbarrow>0 & d⩾30 between *Drawn* and *ArrowReady* states in Fig. 2). Exit actions, on the other hand, can express assignment operations of global variables (*e.g.,* power += d in the gestures linked to the *Bending* state). For model verification we limit the precondition checks to verifying if a global variable is true or false; and exit actions are limited to changes of the boolean value of the global variable. Generally speaking, most of the simplifications on the *GISMO* DSL can be semi-automated and applied to any DSL, thus reducing the task of the domain engineer.

4 Applying *ProMoBox* to *GISMO*

The *ProMoBox* framework [3] relies on a family of fully automated generated modelling languages based on the DSML metamodel. These languages are required to modularly support specification and verification of model properties. The *design language* allows DSM engineers to design the static structure of the system. The *runtime language* enables modellers to define a state of the system, *e.g.,* an initial state as input of a simulation, or a particular "snapshot" during runtime. The *input language* lets the DSM engineer model the behaviour of the system environment, *e.g.,* by modelling an input scenario as an ordered sequence of events containing one or more input elements. The *output language* can be used to represent execution traces (expressed as ordered sequences of states and transitions) of a simulation or to show verification results in the form of a counter-example. Output models can also be created manually as part of an oracle for a test case. The *property language* can be used to express properties based on modal temporal logic, including structural logic and quantification.

A fully automated method specialises and integrates these languages to any given DSML, thus minimising the effort of the language engineer. This is realised by manually annotating the DSML metamodel entities (classes, associations and attributes) with the necessary stereotypes required for every language construct. Stereotype $\ll rt \gg$ (for runtime) annotates metamodel entities that serve as output (*e.g.,* a state variable); stereotype $\ll ev \gg$ (for event) annotates entities that serve as input only (*e.g.,* a marking); stereotype $\ll tr \gg$ (for trigger) annotates static entities that may also serves as input (*e.g.,* a transition event). More stereotypes could be envisioned, but the generation of the sub-languages currently only supports those three. Stereotypes $\ll ev \gg$ and $\ll tr \gg$ cannot be used jointly on the same metamodel entity.

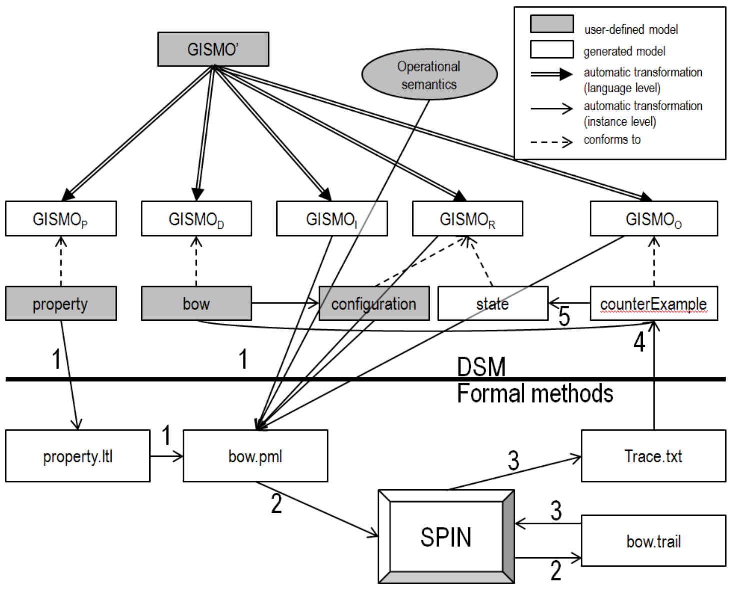


Fig. 3. The *ProMoBox* approach applied to *GISMO*.

Fig. 3 illustrates how to apply *ProMoBox* to *GISMO*. Only the grey parts of Fig. 3 need to be modelled explicitly, the white parts are generated from the annotated and simplified *GISMO*' metamodel, shown in Fig. 1. Classes colored in grey have been removed by the simplification process; stereotype annotations have been added.

A family of languages is generated from *GISMO*' using a template-based approach. This approach makes *ProMoBox* applicable to different types of DSMLs. The main idea is that generic metamodel elements (shown as grey rectangles in Fig. 4) are interwoven with the DSL metamodel elements.

As an example, Fig. 4 shows the metamodel of the generated *output language* $GISMO_O$. The metamodel of generated *design language* $GISMO_D$ closely resembles $GISMO_O$, except that the grey classes in the figure are absent, *OutputElement* is replaced by *DesignElement*, and the *isActive* attribute of *Element* and the *value* attribute of *Bool* is also absent, because of the $\ll rt \gg$ annotations in Fig. 1. The metamodel of generated *runtime language GISMO*$_R$ looks like *GISMO*$_O$, except that the grey classes are absent, and *OutputElement* is replaced by *RuntimeElement*. Fig. 2 shows an example runtime instance model of *GISMO*$_R$, representing a snapshot of the bow model during its execution. The currently active state is displayed in green. The generated metamodel of input language *GISMO*$_I$ is depicted in Fig. 4, together with an example instance model. This model represents a sequence of a *MoveGesture* followed by two *DragGestures* (each gesture is surrounded by a green circle) provided by the user as input and processed by the operational semantics rules to execute the runtime model *GISMO*$_R$. The generated metamodel of *property language GISMO*$_P$ allows to define temporal properties over the system behaviour by means of four constructs: quantification ($\forall$ or $\exists$), temporal patterns, structural patterns and domain-specific pattern elements. These property-related constructs are added to the *GISMO*' metamodel by weaving the generic metamodel template of Fig. 5 into the DSML. The resulting language has a concrete syntax that highly corresponds to the DSML, and the quantification and temporal patterns (instead of LTL operators) are raised to a more intuitive level by using natural language. The full metamodel of *GISMO*$_P$ is not shown here, but a similar example can be found in [3].

Properties specified in *GISMO*$_P$ are translated to LTL, and a Promela model is generated that includes a translation of the initialised system, the environment, and the rule-based operational semantics of the system. This translation is generic, and thus independent of the DSML. The properties are checked by the SPIN model checker. If any counter-example is found, the verification results are translated back to the DSM level.

The limitations of the framework are related to the mapping to Promela as explained in [3]. In its current state, *ProMoBox* does not allow dynamic structure models. Because of the nature of Promela, boundedness is ensured in the translation. Other constraints can be circumvented, as described in Section 3.

5 Specifying and checking properties on GISMO models

We implemented the *ProMoBox* framework in AToMPM [4], and the generic compilers that compile models to and from Promela or text were written in Python. The resulting generated Promela code is around 800 lines of code.

We verified fifteen properties on the runtime instance model of Fig. 2. Five properties are described below:

P_1 It is always possible to return to a previously active state.

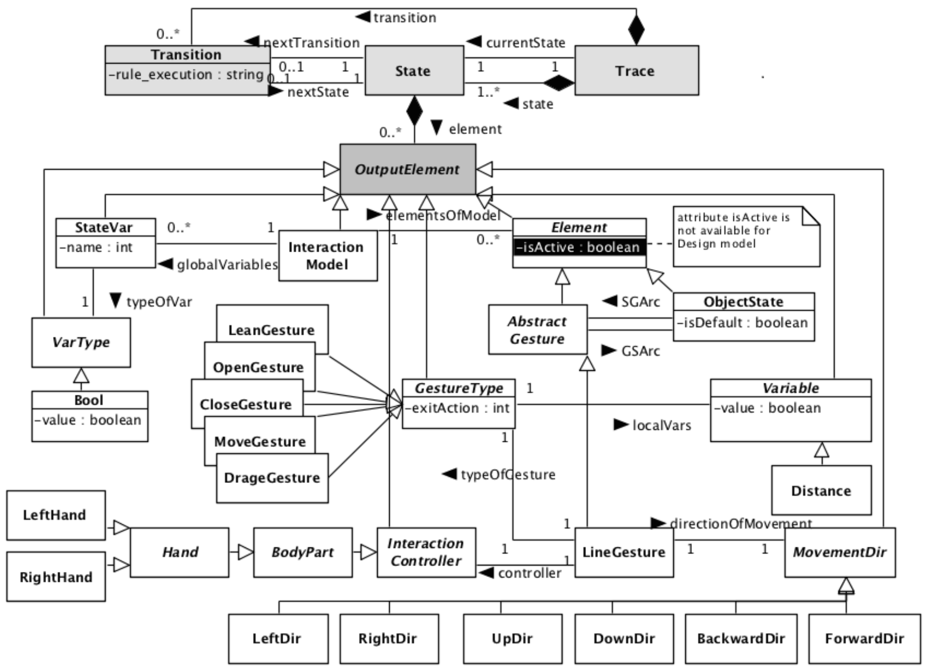
P_2 A bow cannot be bent if there is no arrow on it. (see Fig. 6 at the top left; the global variable *nbarrow* is represented by integer id 1).

P_3 All states of a model can be reached from any state.

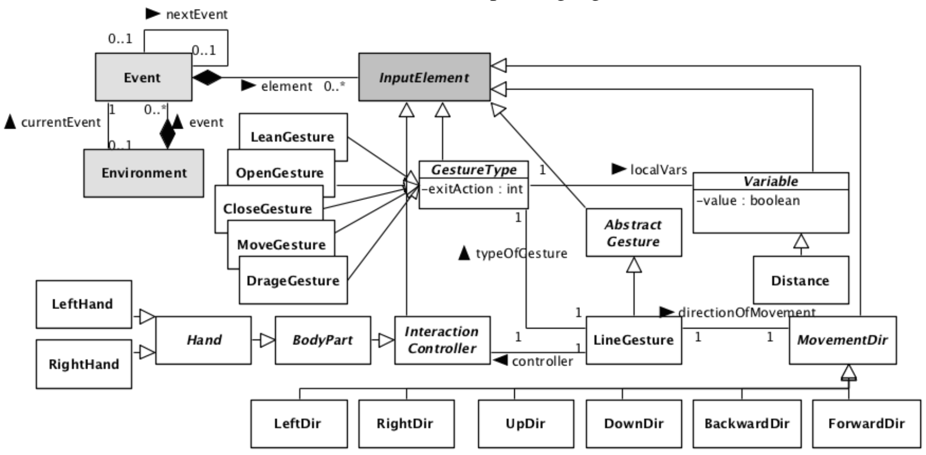
P_4 Whenever the bow is fired, the amount of available arrows should decrement (see Fig. 6 at the top right; the global variable *nbarrow* is represented by integer id 1).

P_5 After firing an arrow, one should eventually be able to fire another one.

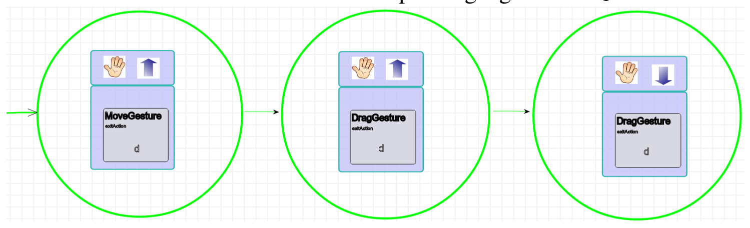
The above properties are transformed to LTL, and are inserted in Promela code consisting of the initial state of the system, the environment and the rule-based operational



Generated metamodel of output language $GISMO_O$



Generated metamodel of input language $GISMO_I$



Example of an instance model of $GISMO_I$

Fig. 4. Generated metamodels $GISMO_O$ and $GISMO_I$, and instance model of $GISMO_I$.

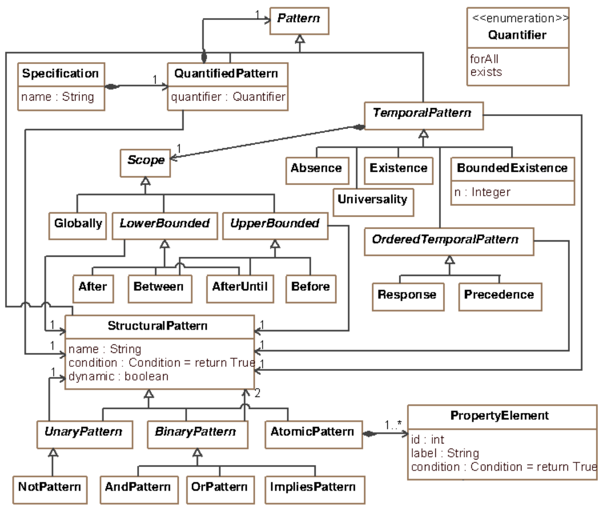


Fig. 5. The generic metamodel template used for generating the property language.

semantics as shown in step 1 of Fig. 3. In step 2, SPIN verifies whether the system satisfies the formula, returning "True" if it does. If there is a counter-example, steps 3 to 5 are followed: the counter-example trace is played back by SPIN, and a readable trace is printed (step 3), this trace is converted automatically to the counter-example output model (step 4), and this counter-example can be played out state by state by showing a runtime model for each state (step 5). The properties P_2 and P_4 yield a counter-example, one of which is shown in Fig. 6 at the bottom. The trace represents a sequence of five states, leading to the undesirable state (the last one, where the state is Bending, but the nbarrow variable has value 0). Each state can be mapped to Fig. 2, and the current state is highlighted, as well as the current input gesture. Because of these counter-examples, we were able to find and fix an error in our bow model of Fig. 2, namely that picking up an arrow (represented by the MoveGesture to the left of sheated) is not required. In another instance, we were able to find and correct an error in one of the operational semantics' rules.

In comparison to [3], we made some changes that influence the performance, such as splitting up quantified rules into several LTL formulae (e.g., Prop. 3). The performance in terms of time and memory consumption is good: evaluation never takes more than a second, and never requires more than 100 MB of memory. Also, the search tree depth never exceeds 4000, and the number of states that are visited stays well under 20000.

6 Related Work

In the last decade, a plethora of language-specific approaches have been presented to specify and verify properties for different kinds of design-oriented languages. A subset of these approaches verify component-based systems [9], concurrent systems [10] and architectural models expressed in UML [11]. Gabmayer *et al.* survey approaches

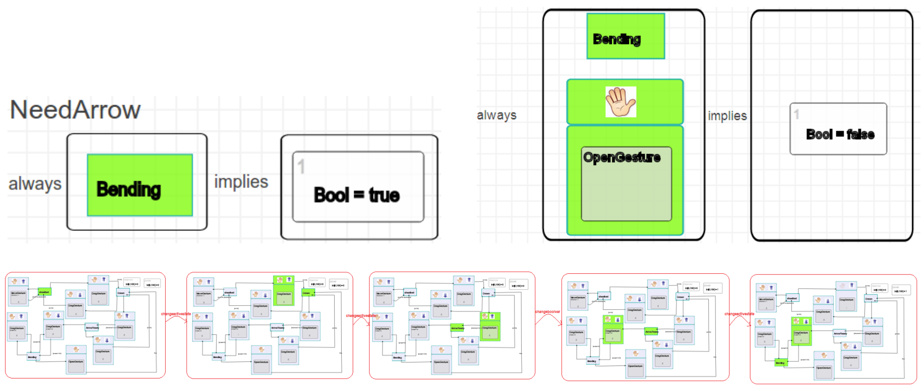


Fig. 6. The two properties P_2 (top left) and P_4 (top right) that yield a counter-example, and the counter-example of P_2 (bottom).

aiming at specifying and verifying temporal model properties by model checkers [12]. These approaches offer either language-specific property languages, or LTL properties have to be defined directly on the formal representation, thus not aiming at supporting DSMLs engineers in the task of building domain-specific property languages.

Generic solutions shift the specification and verification tasks to the model level in a more generalized manner. Some approaches propose OCL extensions for defining temporal properties (TOCL) on models [13,14]. Combemale *et al.* propose a pattern-based extension to modelling languages aiming at supporting temporal property verification using TOCL, while producing model-based input and output automatically for model checking purposes [15]. Klein *et al.* [16] present an approach to define properties at the model level in a generic way, by extending a language for specifying structural patterns based on Story Diagrams with support for specifying temporal patterns.

7 Conclusion and Future Work

This article reported on a practical application of the *ProMoBox* approach [3] for verifying temporal properties on DSMLs. A small number of models is required as input to specify properties and transform them to SPIN, verify them and visualise possible counter-examples, while the user is shielded from the underlying formal model checking intricacies. *GISMO*, a DSML for specifying executable models of gestural interaction applications, was used as a case study. We illustrated how verifying properties on *GISMO* models can be realised with *ProMoBox*, after applying a series of necessary simplifications on the *GISMO* metamodel to ensure that the model is bounded and to avoid a combinatorial explosion of the model checking. We conclude that applying *ProMoBox* to *GISMO* is feasible. Annotating the *GISMO* metamodel to enable the automatic generation of a property language is straightforward. Simplifying the metamodel to enable the model checker to verify properties in a reasonable time took some more effort and reflection. Nevertheless, this is a common step when applying model checking. Lifting the power of model checking to the level of domain-specific models is possible, and the expressiveness of the properties is only limited by the template metamodels of *ProMoBox*, that can be easily extended to include all the concepts of model-checking formalisms such as LTL and Promela.

In future work, we are interested in broadening the types of languages that are supported by *ProMoBox*, e.g. languages and properties that explicitly include time. Also, we will investigate alternative approaches to model checking, such as the generation and execution of test cases, so that the approach becomes more scalable.

References

1. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering. FOSE '07, Washington, DC, USA, IEEE Computer Society (2007) 37–54
2. Risoldi, M.: A methodology for the development of complex domain-specific languages. PhD thesis, University of Geneva (2010)
3. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Wimmer, M., Vangheluwe, H.: ProMoBox: A framework for generating domain-specific property languages. In: Int'l Conf. Software Language Engineering (SLE). (2014)
4. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Mierlo, S.V., Ergin, H.: AToMPM: A Web-based Modeling Environment. In: MoDELS Demonstrations. (2013) 21–25
5. Holzmann, G.J.: The model checker spin. IEEE Trans. Softw. Eng. **23** (1997) 279–295
6. Deshayes, R.: A domain-specific modeling approach for gestural interaction. In: Visual Languages and Human-Centric Computing (VL/HCC). (2013) 181–182
7. Deshayes, R., Mens, T., Palanque, P.: A generic framework for executable gestural interaction models. In: Visual Languages / Human Centric Computing. (2013) 35–38
8. Navarre, D., Palanque, P., Ladry, J.F., Barboni, E.: ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. ACM Trans. Comput.-Hum. Interact. **16** (2009) 18:1–18:56
9. Cimatti, A., Mover, S., Tonetta, S.: Proving and explaining the unfeasibility of message sequence charts for hybrid systems. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design. FMCAD '11 (2011) 54–62
10. Li, X., Hu, J., Bu, L., Zhao, J., Zheng, G.: Consistency checking of concurrent models for scenario-based specifications. In: SDL 2005: Model Driven. Volume 3530 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 298–312
11. Pelliccione, P., Inverardi, P., Muccini, H.: Charmy: A framework for designing and verifying architectural specifications. IEEE Trans. Softw. Eng. **35** (2009) 325–346
12. Gabmeyer, S., Kaufmann, P., Seidl, M.: A classification of model checking-based verification approaches for software models. In: Proceedings of the STAF Workshop on Verification of Model Transformations (VOLT 2013). (2013) 1–7
13. Ziemann, P., Gogolla, M.: Ocl extended with temporal logic. In Broy, M., Zamulin, A., eds.: Perspectives of System Informatics. Volume 2890 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2003) 351–357
14. Bill, R., Gabmeyer, S., Kaufmann, P., Seidl, M.: OCL meets CTL: Towards CTL-Extended OCL Model Checking. In: Proceedings of the MODELS 2013 OCL Workshop. Volume 1092 of CEUR Workshop Proceedings. (2013) 13–22
15. Combemale, B., Crégut, X., Pantel, M.: A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In: Asia-Pacific Softw. Eng. Conf. (APSEC). (2012) 282–287
16. Klein, F., Giese, H.: Joint structural and temporal property specification using timed story scenario diagrams. In: Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering. FASE'07 (2007) 185–199