Assessing Situation Models with a Lightweight Formal Method

Vinicius M. Sobral, João Paulo A. Almeida, Patrícia Dockhorn Costa
Computer Science Department, Federal University of Espírito Santo (UFES)
Vitória-ES, Brazil
vmsobral@inf.ufes.br, jpalmeida@inf.ufes.br, pdcosta@inf.ufes.br

***Abstract*— In order to leverage the benefits of the notion of situation at design time, proper support is required at the modeling level. In the past, this need has led to the development of a situation type specification language called SML. Although SML facilitates the definition of situation types by providing a graphical notation, designers could profit from additional support in order to assess the quality of the situation type models they produce. Since situations consist of combinations of context elements and may also be combined into complex situations, composition may lead to inconsistent, redundant and/or unintended situation type definitions. In order to address this challenge, in this paper we present a formal validation method for situation modeling based on the automatic transformation of SML models into a lightweight formal method.**

Keywords—situation modeling; situation specification; situation validation; situation assessment; model validation;

I. Introduction

The aim of situation-aware applications is to promote effective interaction with users by autonomously adapting application behavior according to the user's current (and projected) situation. When dealing with the design of situation-aware systems we are required to settle a number of questions, including: what are the relevant types of entities that exist in the user's environment (or context)? What are the particular combinations of entities that are relevant to us?

As discussed by Kokar et al. in [1], "*to make use of situation awareness [...] one must be able to recognize situations, [...] associate various properties with particular situations, and communicate descriptions of situations to others*". In order to facilitate those tasks in the development of situation-aware applications, we have been working on techniques and frameworks that explicitly support the *situation concept* at application design time and run-time. Situations are composite entities whose constituents are other context entities, their properties and the relations in which they are involved. Situations support us in conceptualizing certain parts of reality that can be comprehended as a whole. Examples of situations include "John is working", "John has fever", "John is working and has fever", "Bank account number 846-0 is overdrawn while a suspicious transaction is ongoing", etc. The notion of situation enables designers, maintainers and users to abstract from the lower-level entities and properties that stand in a particular situation and to focus on the higher-level patterns that emerge from lower-level entities in time.

We have aimed at simplifying the specification of situation types (at application design time) (see [2], [3]) and the detection of situations (at application run-time) ([4]). At *design-time*, behavior and policies can be defined in terms of the types of situations in which they apply, instead of various low-level conditions. This not only fosters separation of concerns through abstraction but also enables the definition of complex situation types by reusing previously defined situation types. At *run-time*, situation detection machinery can be employed, enabling timely reaction to current situations [4].

In order to leverage the benefits of the notion of situation at design time, proper support is required at the modeling level. This challenge has been addressed in the past with a model-driven approach to the specification of situation types (and ultimately model-driven realization of situation detection) [2]. That approach consists in part of a Situation Modeling Language (SML), which is a graphical language for situation modeling, allowing the expression of primitive situation types and complex situation types (with temporal constraints when required). This means that the designer is able to specify the types of situations in which he/she is interested at a high-level of abstraction, and generate situation detection code automatically.

Despite the benefits of the availability of a suitable modeling language and a code generation infrastructure, the definition of situation types is not a trivial task, and designers could profit from additional support in order to assess the quality of the situation type models they produce. Since situations consist of particular combinations of context elements, their combinations into complex situations may lead to what we call unwanted scenarios, which encompasses inconsistent, redundant and/or unintended situation type definitions.

An *inconsistent situation type definition* specifies an impossible combination of conditions on context elements, and would probably be the result of a design error. A trivial example of inconsistent situation type in a healthcare setting would be a complex situation that is composed of hypothermia and fever simultaneously. Although such inconsistencies may be straightforward to detect, the composition of situations and temporal operators on situations may lead to more subtle relations between situations that may go undetected by the modeler. An inconsistent situation type definition would have no practical purpose for situation-awareness.

Redundant situation type definitions may arise from different forms of specification that actually entail the very same context conditions. Redundant situations would violate parsimony in specifications and have the perverse effect that users would attempt to attribute different semantics to the (apparently) different (yet equivalent) situation types. Consider for example a *fever* situation type, and a *high body temperature* situation type, if both established as sole condition a bodily temperature of 38 degrees Celsius or higher.

Unintended situation type definitions arise out of the difference between the modeler's intention and the actual definitions he/she expresses in the language. This may be a result of lack of knowledge on the semantics of the language or simply the inherent difficulty in predicting all implications of a (complex) definition..

The consequences of those scenarios can vary from excess of situation types to the wrongful detection of situations at runtime (e.g., false positives and false negatives). Most of those problems are related to semantic, domain specific aspects of the situation models, which can only be avoided if a proper model assessment mechanism is employed at design-time.

Model assessment is crucial for the production of high-quality conceptual models in general, and is especially relevant if the models will be employed in a model-driven approach, with the generation of deployable code from models. In our case, we generate situation detection code directly from situation type models, hence the key role of model assessment. Model assessment allows model rectification at an early phase to make the created models less prone to error and to reflect accurately the modeler's intention.

This paper addresses situation type model assessment by proposing the use of a formal method in an approach that is analogous to the one employed in [5] for conceptual models in the Unified Modeling Language (UML). We use a simple but expressive logic-based language called Alloy [6], which is shipped with a sophisticated analyzer. In order to retain the ease-of-use of SML, we automatically translate SML definitions into Alloy, such that the user of the approach is not required to learn this formal method. The Alloy analyzer allows us to check for consistency and redundancy of situation type definitions, as well as to *simulate* those definitions in order to support the modeler in the identification of unintended ones.

This paper is further structured as follows: Section II introduces the SML language, as well as our application scenario in the healthcare domain that will be used in our examples; Section III presents our approach to the assessment problem, the framework used and the transformation from SML to Alloy developed; Section IV discusses the simulation performed, the problems encountered and illustrates with examples in our chosen domain; Section V concludes our work by summarizing the results and proposing topics for further investigation.

II. SML and Running Example

The Situation Modeling Language (SML) is a graphical language for modeling *situation abstractions* in a context-aware (or situation-aware) application scenario. The language was created with the purpose of facilitating the definition of situations types at design-time. SML allows "*the expression of primitive situations and complex situations involving the composition of situations (with temporal constraints when required)*" [2]. A modeling infrastructure for the language was created, and is composed by a graphical editor, which is a model-driven Eclipse plug-in developed with the Obeo Designer tool, and an automatic transformation to a rule-based situation detection platform that leverages on JBoss Drools engine (and its integrated Complex Event Processing platform).

A situation type definition in SML is a composition of two kinds of models: a context model and a situation type model. The context model is a structural model that defines the classes of entities and relationships that exist in the modeled domain, which in turn are referred by the situation type model entities. In order to define context models, we employ an ontologically well-founded UML class diagram profile called OntoUML [7]. OntoUML specializes UML adding important distinctions not originally present in the language; this adds precision to the conceptual context models built with OntoUML. Fig. 1 shows an example of a context model in the *healthcare* domain that incorporates those distinctions, which are explained next, and will be used in our assessment examples. Thus, a class stereotyped as *kind* implies static classification, meaning that this entity (the actual person or hospital in our example) will always be necessarily of that type during its entire life. This is different from a *phase*, which entities play contingently during their lives. In our example, a person will be either infected (here used as a synonym to ill) or healthy, never both, at a specific point in time, but can appear in those phases many times during its life. Also, an infection is always contagious, in order to simplify our simulations. A *relator* (such as treatment in the example) works as a reified association, relating and binding the elements that play *roles* in a relation. Finally a *category* is an abstract class that generalizes different *kinds* (e.g., physical entity).

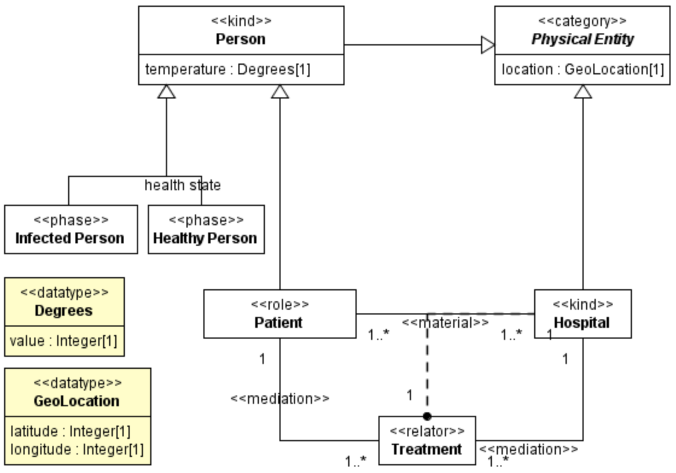


Fig. 1. Healthcare context model

The situation type model defines situation as patterns of the context model classes' instances. SML defines a concrete syntax for creating situation type diagrams, such as the one in

Fig. 2. It illustrates a situation type relevant in the healthcare domain, namely a *fever* situation type, which happens when a person's temperature is above 37 degrees Celsius. The elements depicted in Fig. 2 are references to the homonymous ones created in the context model of Fig. 1. Each situation element that references a context element or another situation is called a *Participant*. For instance, a Participant may be an Entity Participant (reference to any element that is not a relator, such as kind, role, phase, etc.), a Relator Participant or a Situation Participant (reference to another situation). In our example, the Person participant refers to an instance of the kind Person in the context model. More complex situations will be illustrated in section IV.

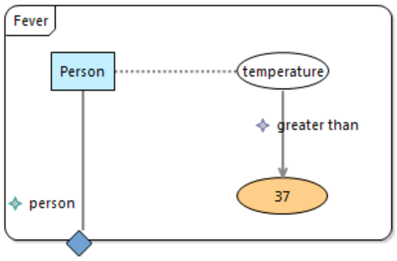
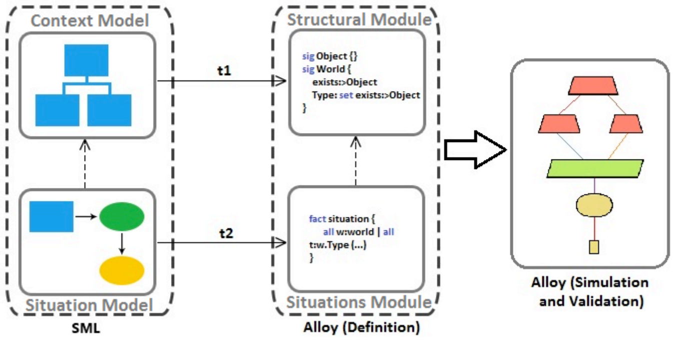


Fig. 2. Example of a SML diagram

III. Model Assessment Approach

A. Approach Overview

The approach comes down to a transformation of our context and situation type models into a logic-based language called Alloy, and further simulation and validation of the obtained models with the Alloy Analyzer tool, as Fig. 3 depicts. The transformation from the context model to Alloy (t1), which we call the *Structural Module*, is inherited from the OntoUML modeling environment, described in the next sub-section. This paper presents the inclusion of a *Situation Module* by transforming the SML situation type model to Alloy (t2). The situation module depends on the structural module as the situation type model depends on the context model. Having both modules in the Alloy definition allow us to simulate with the analyzer *worlds* populated with context objects and situation instances to assess our models. We can also check the validity of assertions that quantify over the states and histories of context objects and situation instances. The Alloy specification is basically a set of *facts* which represent *axioms* that are respected in the simulation and validation.



```
sig Object {}
sig World {
  exists:>Object
  Type: set exists:>Object
}
```

```
fact situation {
    all w:world | all
 t:w.Type (...)
 }
```

Fig. 3. Approach Overview

B. The Modeling Environment and Validation Framework

The OntoUML language (used for our context models) has a rich model-based environment for model construction, verbalization, code generation, formal verification and validation. Our assessment approach makes use, in particular, of the validation framework [5] (which uses the Alloy language, present in this environment), incorporating a *Situation Module* to it.

Alloy [6] is a logic language based on set theory, which is supported by an Analyzer that, given a context, exhaustively generates possible instances for a given specification and also allows automatic checking of assertions' consistency. In the OntoUML validation framework, the generated instances of a given conceptual model are organized in a branching-time temporal structure, thus, serving as a visual simulator for the possible dynamics of element creation, classification, association and destruction. It allows a modeler to visualize a representation of snapshots in this world structure, which are states admissible by the models current axiomatization. This enables modelers to detect unintended, redundant or inconsistent model instances and take the proper measures to rectify them.

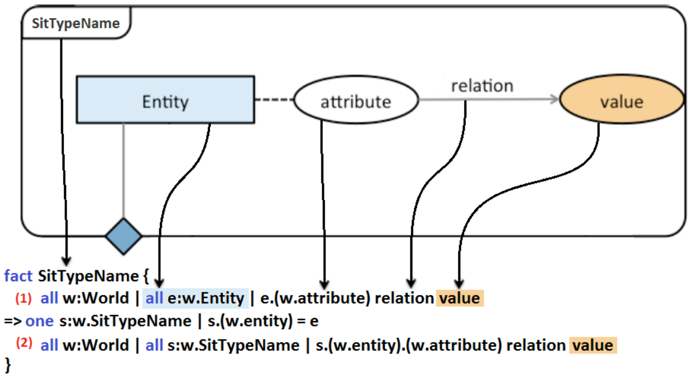
C. The Transformation

The transformation from SML to Alloy was conceived in a model-driven fashion. Each concept of the SML situation type metamodel was mapped to a respective *pattern* in Alloy and the union of these patterns represents what we call the situation axiom. We create the situation axioms in Alloy accordingly to the formalization of SML presented in [2] (in first-order logic).

Each situation axiom postulates the conditions for the existence of a situation of a particular type, i.e., those conditions that must be true for as long as the situation of the type exists. In Alloy we address this with a fact with two expressions: one that captures the <u>sufficient</u> conditions for the existence of the situation (which necessitates the creation of a situation of the type using the =>/implies operator) and one that captures the <u>necessary</u> conditions. Since these facts are specific to a particular situation type, we present transformation rules that determine these facts from situation type definitions in SML.

The most important rules in the transformation are depicted in the following figures. The figures show an element of a situation model and its respective representation in Alloy connected by an arrow. For every situation type example, the first rule in a fact represents the sufficient conditions (marked with "1") and the second rule the necessary conditions (marked with "2"). The explanation of the transformation patterns will focus on former, since the latter is analogous.

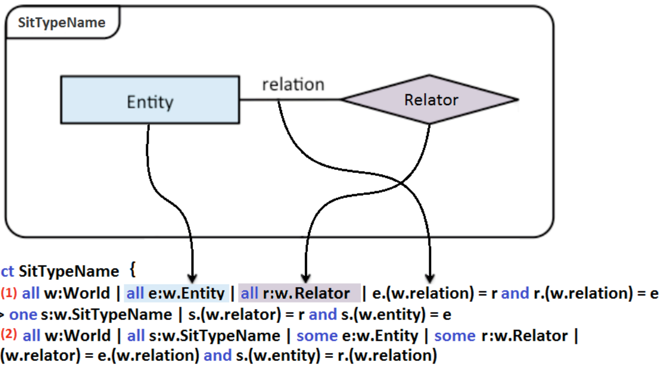
Fig. 4 shows the transformation of Entity Participants (depicted only as Entity), Attributes and Formal Relations. In this example one can see that a situation always generates a fact with equivalent name. Likewise, every rule begins by universally quantifying the worlds which we will talk about (in this case only one World *w*). *Participants* (entities, relators and other situation references) will always be quantified also (like the entity in the example). After the quantification of all elements we apply the restrictions that compose the situation rule, which can be world relationships, formal relations such as the one depicted (an *attribute* of *e* related to a *value* by a relation *relation*) or other associations. At the end of the rule we have a pattern that indicates that those restrictions imply the existence of a situation *s* of type *SitTypeName* and that this situation is bound to the instance *e* (would also bind every other *Participant*, if it was the case). The entity's name in lowercase is used to represent the association between the situation and the entity that must be created in alloy. The equality at the end of the rule indicates that the entity connected to the situation in the world at issue must necessarily be *e*.



```
fact SitTypeName {
  (1) all w:World | all e:w.Entity | e.(w.attribute) relation value
=> one s:w.SitTypeName | s.(w.entity) = e
  (2) all w:World | all s:w.SitTypeName | s.(w.entity).(w.attribute) relation value
}
```

Fig. 4. Transformation rule for Entities, Attributes and Formal Relations

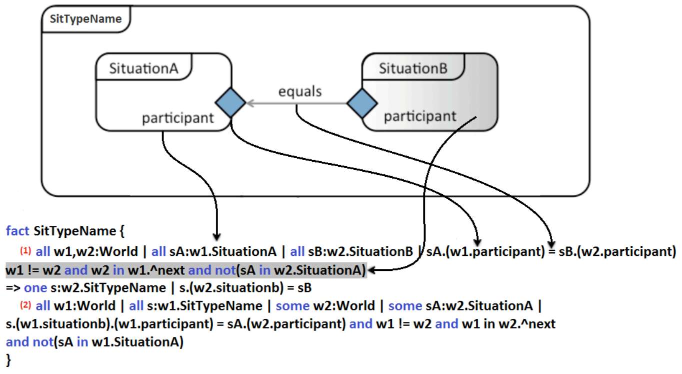
Fig. 5 depicts the transformations rule for Relational Contexts. In this case, both the Entity and the Relator are quantified. The relation between them is transformed into the restriction that triggers the situation (the object that connects to *e* through the relation *relation* must be *r* and vice versa). At the end, the binding is made both for the Entity and the Relator (in truth, only one side of the restriction and one binding would be necessary, but we opted to represent the complete rule).



```
fact SitTypeName  {
  (1) all w:World | all e:w.Entity | all r:w.Relator  | e.(w.relation) = r and r.(w.relation) = e
=> one s:w.SitTypeName | s.(w.relator) = r and s.(w.entity) = e
  (2) all w:World | all s:w.SitTypeName | some e:w.Entity | some r:w.Relator |
s.(w.relator) = e.(w.relation) and s.(w.entity) = r.(w.relation)
}
```

Fig. 5. Transformation rule for Relational Contexts

Fig. 6 shows the transformation of situation composition. Past (white rounded rectangle) and current (shaded rounded rectangle) situations are represented in Alloy by means of reified worlds. Each past world in the situation definition adds a new world variable to the rule. World relations are defined by the *next* operator, so a past world is actually a world of which the set of future (next) worlds contains the current world. Past situations like SituationA are constrained to exist in the past world (*w1* in (1) and *w2* in (2)) and not exist in the current world, while current situations exist in the same world the situation type being defined exists. Finally, the *equals* relation becomes an equivalence in Alloy.



```
fact SitTypeName {
  (1) all w1,w2:World | all sA:w1.SituationA | all sB:w2.SituationB | sA.(w1.participant) = sB.(w2.participant)
w1 != w2 and w2 in w1.^next and not(sA in w2.SituationA)
=> one s:w2.SitTypeName | s.(w2.situationb) = sB
  (2) all w1:World | all s:w1.SitTypeName | some w2:World | some sA:w2.SituationA |
s.(w1.situationb).(w1.participant) = sA.(w2.participant) and w1 != w2 and w1 in w2.^next
and not(sA in w1.SituationA)
}
```

Fig. 6. Transformation rule for Situation composition

In addition to the situation type specific facts that we have discussed so far, we must also admit an axiom that states that *a situation is unique for a particular conjunction of entities in a world.* This is a general axiom that is represented as a predicate in Alloy and must be applied to every situation type defined:

```
// Situation Uniqueness
pred situationUniq[sit: univ->univ, essentpart:univ->univ->univ] {
   all w:World | all s1,s2:w.sit | s1.(w.essentpart) = s2.(w.essentpart) implies s1 = s2
}
```

Finally, we must also admit for every situation type that *if a conjunction of entities remains in a particular condition in two consecutive worlds, then the situations in both worlds are the same*. Again, this is a general axiom that is represented as a epredicate in Alloy and must be applied to every situation type defined:

```
//Situation Continuity
pred situationCont[sit: univ->univ, essentpart:univ->univ->univ] {
   all w1,w2:World | all s1: w1.sit, s2: w2.sit | w2 in (w1.next) and
s1.(w1.essentpart) = s2.(w2.essentpart)
implies s1 = s2
   all w1,w2:World | all s1: w1.sit, s2: w2.sit | w2 in (w1.next) and s1 = s2
implies s1.(w1.essentpart) = s2.(w2.essentpart)
}
```

IV. Example Assessment

We start the illustration of model assessment with a simple situation. Take, for example, the fever situation of Fig. 2, which exists when a person's temperature is above 37 degrees Celsius. After transforming the situation model to Alloy (along with the context model), we obtain a rule analogous to the generic ones shown in the last section. At this point we can ask Alloy to generate worlds so that we can check for under and overconstraining, redundancy and inconsistency. A very simple Alloy simulation with our fever situation is shown in Fig. 7, which shows a *Fever* situation which involves an Object of type *Person* (also *Infected Person* and *Physical Entity*) that has a temperature of 40 Degrees Celsius.

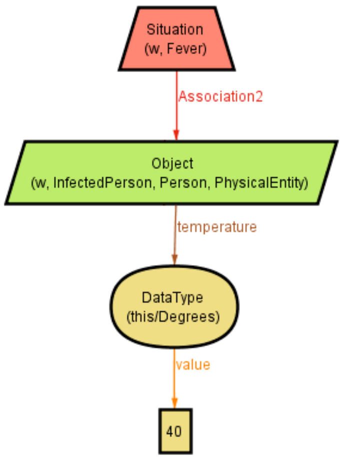


Fig. 7. Simulation of a Fever situation in Alloy

Alloy gives us a possibility of checking our formal relations since SML does not constrain their use. For instance, the incorrect use of temporal relations between composing situations in a complex situation can generate unsatisfiable situations. The situation in Fig. 8 illustrates a past situation that occurs in a time interval from zero to two hours after a current situation, which is impossible.

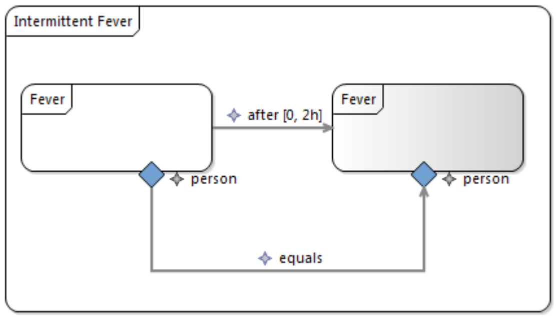
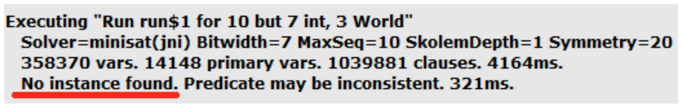


Fig. 8. Intermittent Fever situation with wrong temporal relation

Thus, in Alloy, we will have a rule that says one situation is both in a world that is next and previous to the current world. This rule, as we would expect, is unsatisfiable, as we can check by applying the following rule to the run command in Alloy, which asks the analyzer to generate at least one instance of the *Intermittent Fever* situation:

```
run {
some w:World|#w.IntermittentFever>=1
} for 10 but 3 World, 7 int
```

After checking exhaustively all the possibilities within the defined scope, the tool present us the message depicted in Fig. 9. It says that the predicate may be inconsistent for we have limited the number of instances it should generate, but we can easily see that no bigger scope is necessary since the number we used would be sufficient if there were to exist the situation.



```
Executing "Run run$1 for 10 but 7 int, 3 World"
  Solver=minisat(jni) Bitwidth=7 MaxSeq=10 SkolemDepth=1 Symmetry=20
  358370 vars. 14148 primary vars. 1039881 clauses. 4164ms.
  No instance found. Predicate may be inconsistent. 321ms.
```

Fig. 9. No instance found for the command run. Unsatisfiable result.

Another unsatisfiable situation can occur when we create a composite situation with contradicting situation conditions. For instance, suppose a situation where a person's temperature is above 40 degrees Celsius and also below this same temperature. This is clearly a wrong definition but one that can be difficult to identify if we have a model with many situation types, or if it is masked by the situation's names. An example of this is shown in Fig. 10, where the user was misled by the situation name to think that *Under 40* refers to the person's age instead, while a *High Fever* situation is defined as a person that has more than 40 degrees Celsius. After generating the Alloy rule, we run a command analogous to the one for *Intermittent Fever*, but asking Alloy to generate at least one instance of an *Under 40 High Fever* situation instead. As we would expect, no instance was found, indicated that this situation is inconsistent.

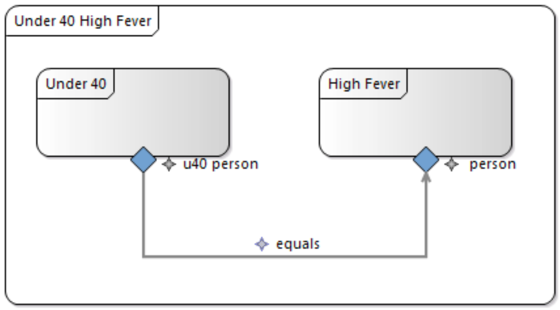


Fig. 10. Situation composed by two incompatible situations

The Alloy Analyzer can also be used to check equivalence of situations. In a large model, many types can be created to indicate a same situation, which is undesired since it overpopulates the model and do not add semantics to it. For instance, we have a *Normal Fever* situation (Fig. 11), which is defined as a person who has a temperature between 37 and 40 degrees Celsius, and a *Common Fever* situation (Fig. 12), which is the composition of the *Fever* (temperature > 37ºC) and *Under 40 C* (temperature < 40ºC) situations definitions.

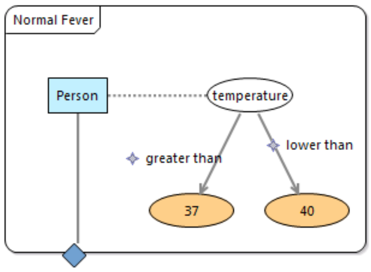


Fig. 11. Normal Fever situation

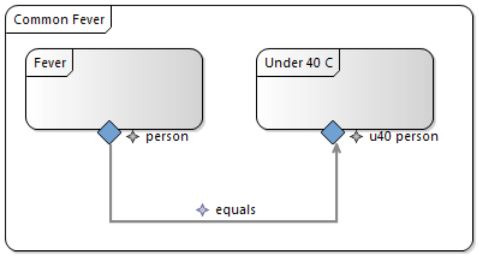


Fig. 12. Common Fever situation

Both of them should happen at the same temperature interval and, consequently, at the same time, as we see by running the simulation. Every world generated by the analyzer is similar to the one in Fig. 13, where there is a *Common Fever* situation (Situation3), connected (composed by) to a *Fever* and an *Under 40 C* situation, always alongside a *Normal Fever* situation (Situation1), the two referring to the same person.

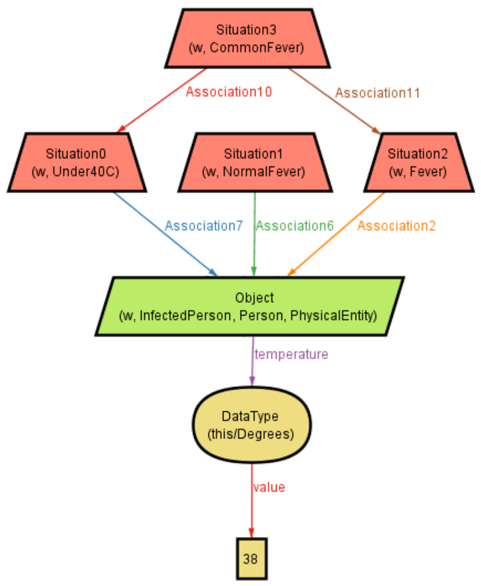
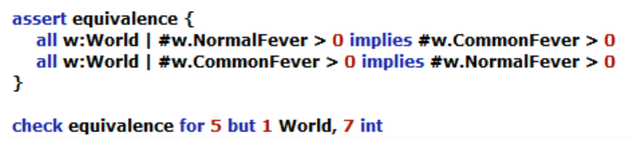


Fig. 13. Alloy simulation for Common Fever and Normal Fever

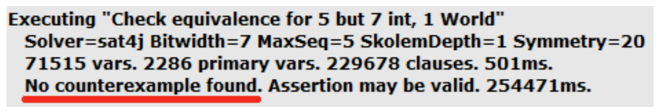
As a definitive test for this case, we can create assertions and ask Alloy to verify whether it holds. An assertion is an assumed true state which the analyzer tries to contradict. A successful contradiction means a false assertion, and is supported by an example to the user, while an unsuccessful one mean the assertion is true for the scope we defined. Consequently, we created an assertion that affirms that whenever a *Common Fever* (the number of its instances is > 0) exists, a *Normal Fever* also exists and vice versa (depicted in Fig. 14). One can check the result obtained, which indicates that the assertion is valid, in Fig. 15.



```
assert equivalence {
   all w:World | #w.NormalFever > 0 implies #w.CommonFever > 0
   all w:World | #w.CommonFever > 0 implies #w.NormalFever > 0
}

check equivalence for 5 but 1 World, 7 int
```

Fig. 14. Assertion for checking equivalence of situations



```
Executing "Check equivalence for 5 but 7 int, 1 World"
  Solver=sat4j Bitwidth=7 MaxSeq=5 SkolemDepth=1 Symmetry=20
  71515 vars. 2286 primary vars. 229678 clauses. 501ms.
  No counterexample found. Assertion may be valid. 254471ms.
```

Fig. 15. No counterexample found. Valid assertion.

Finally, we can use the analyzer to detected unintended situation type definitions resulting from underconstrained models. Suppose, first, we have the following situations (Fig. 16): a person is healthy (*Healthy* situation); a person is infected (*Infected* situation); a healthy person becomes infected (*Becomes Infected* situation); and a *Patient* is having a treatment in a *Hospital* (*Is Being Treated* situation). We want now to combine the situations presented to create a more complex situation in which two *Patients*, one which is also an *Infected Person*, are being treated in a *Hospital* at the very same time (overlapping *Is Being Treated* situations), and the *Patient* which was healthy becomes infected (*Becomes Infected* situation) after the treatment. This characterizes our *Possible Contagion* situation, whose model can be seen in Fig. 17.

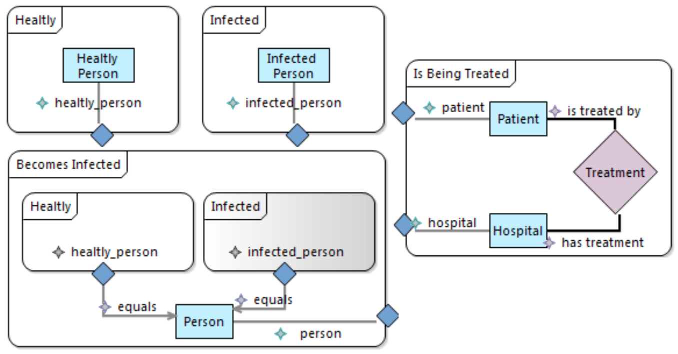


Fig. 16. Becomes Infected and Is Being Treated situations.

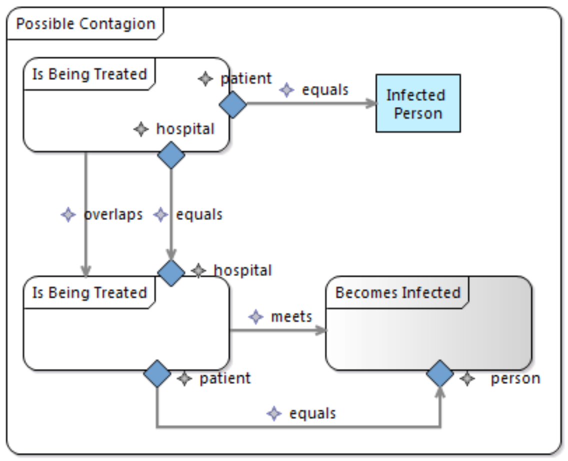


Fig. 17. Possible Contagion situation.

As we ask alloy to generate instances of the modeled situation, we see in Fig. 18 that we have underconstrained our definition since it is feasible that the same person alone (Object1) generates a *Possible Contagion* situation (Situation2) by turning from an *Infected* to a *Healthy Person* (indicated in the first two worlds of the image) and becoming *Infected* (Situation3) in the subsequent world.

This situation happens for two reasons: a *Patient* can have multiple treatments in the same hospital at the same time (defined by our context model, which is reasonable) and thus can be the Patient of both *Is Being Treated* situations. Besides, we didn't explicitly said the Patient from the first treatment must be different from the Patient from the second treatment (and consequently from the infected Person). A correct situation model for this case would be the one in Fig. 19, in which we added the negated (not) *equals* relation between the person originally infected and the one that becomes so.

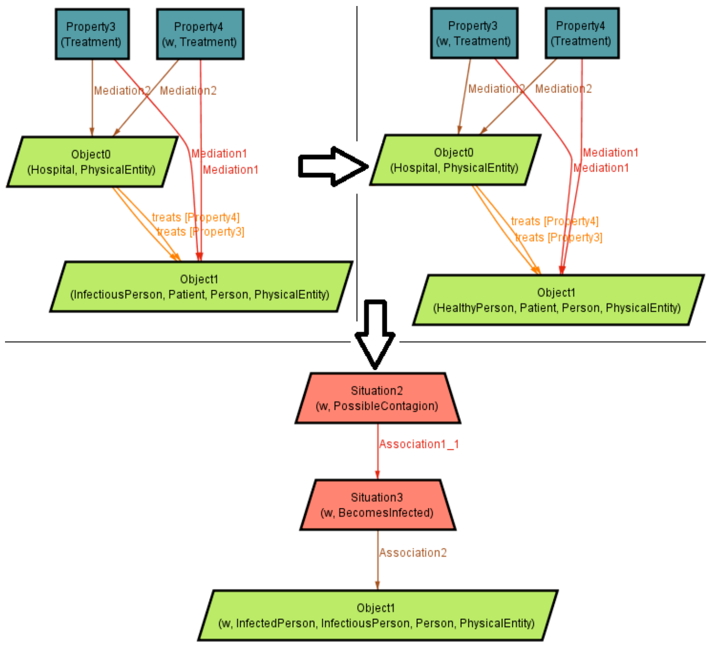

Property3 (Treatment)
Property4 (w, Treatment)
Mediation2
Mediation2
Mediation1
Mediation1
Object0 (Hospital, PhysicalEntity)
treats [Property4]
treats [Property3]
Object1 (InfectiousPerson, Patient, Person, PhysicalEntity)
Property3 (w, Treatment)
Property4 (Treatment)
Object0 (Hospital, PhysicalEntity)
Object1 (HealthyPerson, Patient, Person, PhysicalEntity)
Situation2 (w, PossibleContagion)
Association1_1
Situation3 (w, BecomesInfected)
Association2
Object1 (w, InfectedPerson, InfectiousPerson, Person, PhysicalEntity)

Fig. 18. Epidemic Risk situation generation by one instance alone

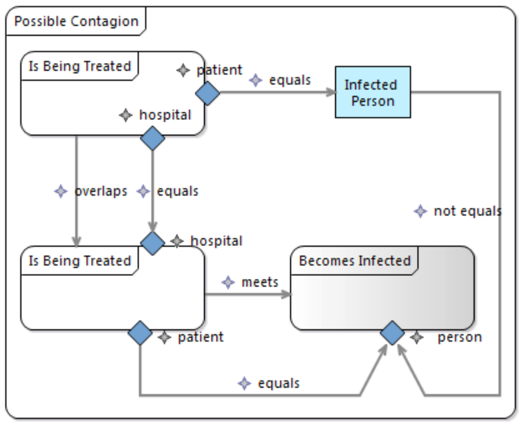


Fig. 19. Correct Possible Contagion situation

V. Conclusions and Future Work.

This paper proposes an assessment approach for situation models using a lightweight formal method. In order to accomplish that in a manner that is transparent to the user, we have proposed an automated transformation from SML to Alloy and analysis of the result using the Alloy Analyzer. As we have demonstrated, our approach allows us to identify from simple inconsistencies (solved by adding a single element or constraint), to more sophisticated semantic and equivalence problems, which are hard to notice without the help of an automated tool.

Our approach uses the validation framework developed in [5], but also extends it including a *Situation Module* that enables one to validate situation type models developed with SML. Although situation modeling is a recurrent subject in the context-aware applications community, situation model validation/assessment at design time is applied only for very specific scenarios and technologies such as in [8], unlike the more general domain-independent approach we take in this paper. Meanwhile, conceptual model validation is more recurrent and OntoUML, along with its validation framework, was used in [9] to detect semantic anti-patterns in this language (although not including support for situations).

For future work we intend to continue exploring the SML language and increasing its expressivity and its infrastructure. In particular, we intend to expand the language's constructs to be able to express the subtleties of the acquisition of context information (including quality of context). This will require extending the approach we present in this paper. We also intend to improve the visualization of situation instantiation for assessing situation models. Although we have used so far the visualization tool provided with the Alloy Analyzer, we believe that a richer tool with explicit support for the situation concept may be more appropriate, allowing us to explore richer graphical patterns. Diagrams generated by the Analyzer would then be used to communicate with domain experts. Finally, we will continue to work on our approach aiming to propose a systematic method for validation of situation type models.

Acknowledgements

This research is funded by the Brazilian Research Funding Agencies FAPES (grant number 59971509/12), CNPq (310634/2011-3, 485368/2013-7) and CAPES.

References

[1] M. M. Kokar, C. J. Matheus, and K. Baclawski, "Ontology-based situation awareness," *Inf. Fusion*, vol. 10, no. 1, pp. 83–98, Jan. 2009.

[2] P. D. Costa, I. T. Mielke, I. Pereira and J. P. A. Almeida, "A Model-Driven Approach to Situations: Situation Modeling and Rule-Based Situation Detection," in *2012 IEEE 16th International Enterprise Distributed Object Computing Conference, 2012, Beijing, China, Sept. 10-14, 2012*, no. 87346, pp. 154–163.

[3] P. Costa, "Architectural Support for Context-Aware Applications: From Context Models to Services Platforms," Universiteit Twente, Enschede, The Netherlands, 2007.

[4] I. S. A. Pereira, P. D. Costa, and J. P. A. Almeida, "A rule-based platform for situation management," in *2013 IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support (CogSIMA)*, 2013, pp. 83–90.

[5] A. B. Benevides, G. Guizzardi, B. F. B. Braga, and J. P. A. Almeida, "Validating Modal Aspects of OntoUML Conceptual Models Using Automatically Generated Visual World Structures.," *J. UCS*, vol. 16, no. 20, pp. 2904–2933, 2010.

[6] D. Jackson, *Software Abstractions - Logic, Language, and Analysis.* MIT Press, 2006, pp. I–XVI, 1–350.

[7] G. Guizzardi, "Ontological foundations for structural conceptual models," CTIT, Centre for Telematics and Information Technology, Enschede, 2005.

[8] S. Lu, A. Tazin, and M. M. Kokar, "Network composition for situation assessment: A trusted meeting case study," in *Information Fusion (FUSION), 2012 15th International Conference on*, 2012, pp. 346–353.

[9] G. Guizzardi and T. P. Sales, "Detection, Simulation and Elimination of Semantic Anti-patterns in Ontology-Driven Conceptual Models," in *Conceptual Modeling - 33rd International Conference, 2014, Atlanta, GA, USA, October 27-29, 2014. Proceedings*, 2014, vol. 8824, pp. 363–376.