Safety and Security Assessment of Behavioral Properties Using Alloy

Julien Brunel, David Chemouil

▶ **To cite this version:**

Julien Brunel, David Chemouil. Safety and Security Assessment of Behavioral Properties Using Alloy. 2nd International workshop on the Integration of Safety and Security Engineering, Sep 2015, Delft, Netherlands. 10.1007/978-3-319-24249-1_22 . hal-01206638

HAL Id: hal-01206638
https://hal.science/hal-01206638

Submitted on 29 Sep 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Safety and Security Assessment of Behavioral Properties Using Alloy

Julien Brunel, David Chemouil

ONERA-DTIM
2 Av Edouard Belin, BP 74025, F-31055 Toulouse
Firstname.Lastname@onera.fr

Abstract. In this paper, we propose a formal approach to supporting safety and security engineering, in the spirit of Model-Based Safety Assessment, using the Alloy language. We first implement a system modeling framework, called Coy, allowing to model system architectures and their behavior with respect to component failures. Then we illustrate the use of Coy by defining a fire detection system example and analyzing some safety and security requirements. An interesting aspect of this approach lies in the "declarative" style provided by Alloy, which allows the lean specification of both the model and its properties.

Keywords: Logic, Formal Methods, Alloy, Safety and Security

1 Introduction

In the context of critical systems engineering, formal approaches have been used for a long time with great successes. In particular, in order to support safety analyses, an approach called Model-Based Safety Assessment (MBSA) [3] has been proposed. The language AltaRica [1] and associated tools is one the main techniques used in MBSA. It allows to model system architectures as a set of communicating automata (one automaton per function or system, depending on the level of abstraction retained for the system under study) and then to study the impact of failure or erroneous events. Then, for instance, fault trees may be generated, the impact of events can be simulated or some property can be assessed exhaustively using a model-checker. Besides, some security properties can also be addressed with the same kind of approach as the language is in fact agnostic with respect to the nature of feared events.

Following earlier work, we propose here to address the question of safety and security assessment using the Alloy language and the Alloy Analyzer free-software tool. Alloy [9] is a formal modeling language amenable to automatic analyses. Alloy has recently been used in the context of security assessment, for instance to model JVM security constraints [11], access control policies [12], or attacks in cryptographic protocols [10]. Besides, we proposed in earlier works a study of the safety and security assessment of an avionic system supporting an approach procedure [4,5,6].

Our motivation for relying on Alloy instead of, say, AltaRica is to take benefit from the model-based aspect of Alloy and its expressiveness for the specification of the properties to check. Indeed, Alloy allows to define metamodels easily, which allows for

instance to devise domain-specific metamodels. Here, as will be seen, we develop in Alloy a modeling framework called Coy which can be partly seen as the embedding of the general concepts of AltaRica into Alloy (ignoring concepts we do not need). Furthermore, with Alloy, the specification of the properties we check is expressed in relational first-order logic, with many features adapted to model-based reasoning.

With respect to our previous propositions around using Alloy for MBS&SA, we devise here a richer architectural framework and, more importantly, we formalize a notion of behavior so as to be able to check properties of the considered system along time.

Thus, this paper is organized as follows: in Sect. 2, we give a very brief account of Alloy. Then, in Sect. 3, we describe the Coy modeling framework that we implemented in Alloy to model system architectures and their behavior. We show how Alloy is well adapted to designing domain-specific metamodels and to getting some flexibility in the modeling of time and behavior. In Sect. 4, we illustrate our approach on a fire detection example that we model in Alloy following the Coy metamodel. In particular, we show how using Alloy allows to express "in one shot" properties ranging over a set of elements selected by navigating in the model structure.

2 Alloy in a Nutshell

Alloy is a formal modeling language that is well adapted to the following (non-exhaustive) list of activities: abstract modeling of a problem or of a system; production of a metamodel (model corresponding to a viewpoint); analysis of a model using well-formedness or formal semantic rules; automatic generation of an instance conforming to a model, possibly according to supplementary constraints; finding interesting instances of a model. Models designed in Alloy can deal with static aspects only, or integrate also dynamic aspects, so as to check behavioral properties.

We now give a brief glance at the main concepts of the language using a simple example.The most important type of declaration is that of a *signature* which introduces a structural concept. It may be seen as a class or entity in modeling parlance. A signature is interpreted as a set of possible instances; and it can also come with fields that may be seen, as a first approximation, as class attributes or associations.

```
sig Data { consumedBy : some System }
sig System {}
sig Criticality {
  concernedData : one Data,
  concernedSystem : one System
}
```

Here, we defined 3 concepts : Data, System and Criticality. Alloy advocates not to delve into unnecessary details and only give information on things we want to understand or analyze. Thus, here, a system is just defined to be a set of "things", but we do not say anything about the exact nature of its elements.

The keywords **some** or **one** give details on the multiplicity of the relation, as `1..*` and `1` in UML. Here the field declarations mean that: every datum is consumed by

at least one (some) system; every criticality concerns exactly one (one) data and one system. Other possible multiplicities are: **lone** which means at most one (0..1); and **set** which means any number (0..*).

Then, we can add constraints on possible instances of our model. For instance, we would like to state that every system consumes at least one datum. This can be done by writing additional facts (facts are axioms, so as few facts as possible should be stated in order to avoid over-specification):

```
fact {
  // every system consumes at least one datum
  all s : System | some consumedBy.s
  // for any system which consumes a given datum, the said datum and system
  // should belong to a same unique criticality
  all d : Data | all s : System | one c : Criticality |
    c.concernedData = d and c.concernedSystem = s
}
```

The . operator yields the join of two relations, matching the last column from the first one to the first column of the second one. Thus one may write d.consumedBy to get the systems consuming a data d, but also consumedBy.s to get the data consumed by the system s.

The formal foundation of Alloy is relational first order-logic, that is first-order logic extended with relational terms (including the transitive closure of a binary relation). Besides allowing navigation in models, this logic suffices to encode various models of time (*e.g.* to go from a linear to a tree view of time, or to give either an interleaving or a true-concurrency semantics).

Finally, although the language does not preclude unbounded verification in principle, in practice the Alloy Analyzer works only on *finite* models, reducing a given problem to a SAT instance the analysis of which is delegated to an off-the-shelf SAT solver. Then Alloy may be used to carry out some explorations (the command **run** builds instances that satisfy a given statement) or to check whether a given *assertion* is satisfied by all instances of the model (command **check**). Therefore, as analysis is sound but carried out on finite instances only, the Alloy Analyzer is able to find counter-examples *up to a certain bound* but it cannot prove the validity of an assertion. This is not a problem in our case because (1) the system architecture we consider is fixed in advance so its number of instances may not vary and (2) only time (*i.e.* the size of the time model) may be unbounded but, in our analyses, we do not aim at proving the absence of errors but rather that a bounded number of events does not lead to a feared situation (which induces that bounded time is sufficient).

3 The Coy Modeling Framework

We now present the Coy modeling framework, implemented as a metamodel in Alloy (*i.e.* a model where each signature is abstract and only instantiated in a second model corresponding to the system under study). We take inspiration in model-based safety assessment but our formalization is not specific to this sole family of properties.

As will be seen hereafter, Coy models essentially represent hierarchical structures of transition systems communicating instantaneously through data ports.

The overall structure of the framework is presented graphically in Fig. 1. Extension links are figured using black dashed arrows. As the metamodel contains n-ary relations with $n > 2$, the figure shows these after projection on parts of their domain (this is indicated using square brackets, as in `conns[Port]` for instance). Furthermore, the metamodel contains a `Time` signature: its purpose is that every signature field with `Time` as its last column can be conceptually seen as mutable field, *i.e.* its value may change (discretely) over time. Notice that the metamodel in Fig. 1 is projected over `Time`, hence it is not shown in the diagram.

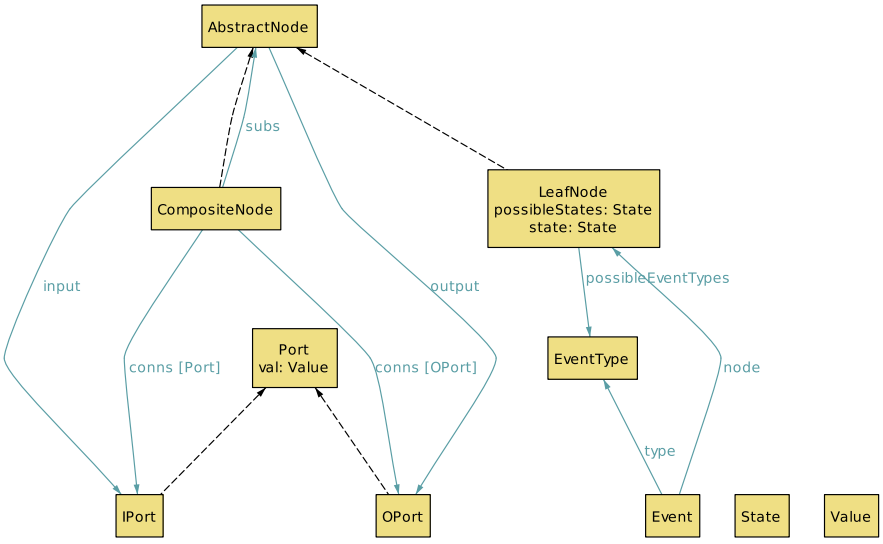


Fig. 1: Graphical depiction of the Coy metamodel (projected over `Time`)

3.1 Composite Structure

Let us now delve into more details in the metamodel (in what follows, for the sake of readability, we do not show all Alloy facts enforcing the well-formedness of instance models or just classical properties). The basic architectural element is a *node*. Nodes are arranged hierarchically as a tree, so we use the classical *Composite* design pattern and devise a notion of `AbstractNode` which is inherited by signatures `CompositeNode` and `LeafNode`, the former pointing back to abstract nodes.

Every node comes with a set `IPort` of input ports and a set `OPort` of output ports. These sets are disjoint and every port belongs to a single node. Every port carries a value at every instant (possible values may differ for distinct ports).

Connections (between ports) are constrained so that they cannot cross a parent node boundary or many levels of composition. In other words, nodes are arranged as trees

and connections can only happen between siblings or between a parent and a child. Furthermore, connected ports always carry the same value.

```
abstract sig Port { // a port carries one value at every instant
  val : Value one →Time }
abstract sig IPort, OPort extends Port {}
abstract sig AbstractNode { // input and output ports
  input : set IPort,
  output : set OPort }
abstract sig CompositeNode extends AbstractNode {
  // a composite node contains at least one sub−node
  subs : some AbstractNode,
  // port connections with siblings and between sub−nodes and this node
  conns : subs.@output →subs.@input +input →subs.@input
          +subs.@output →output,
} { // connected ports always carry the same value
  all t : Time, po, pi : Port | po→pi in conns implies po.val.t = pi.val.t
  // +other structural properties
  ... }
abstract sig LeafNode extends AbstractNode { ... }
```

3.2 Behavior

As Coy is mainly aimed at describing systems where atomic nodes are endowed with behavior, we now introduce a notion of state (for leaf nodes) and of events that may happen. One approach to deal with such models could be to rely on classical model-checkers, such as Spin [8] or NuSMV [7], the modeling languages of which are well-suited to describing transition systems. While this is of course a possibility, our aim with using Alloy is:

- to be able to easily adapt the Coy metamodel depending on the domain of study (*e.g.* to add a notion of connectors as in many architecture-description languages);
- as explained earlier, to change the model of time if need be (*e.g.* to go from a linear to a tree view of time, or to give either an interleaving or a true-concurrency semantics);
- and above all to allow the use of logic for specification, which brings two interesting aspects:
  1. it provides a single language to specify both models and expected properties;
  2. the logic allows for the expression of rather abstract properties (*e.g.* relying on under-specification) or to navigate through elements of a model to specify a given property in only one formula.

Thus, every leaf node is in one given state at any time (the set of possible states may vary for two different nodes). Besides, such nodes may undergo events. An event is an instance of an event type that happens at a certain instant and concerns a given node: this distinction between events and event types then allows to consider events of a certain type only, for instance to characterize their effects.

Notice also we impose the end-user to give, for any leaf node, the set of its possible states and the set of event types that concern it: this is a bit redundant from the theoretical point of view but it provides a sort of additional safety check akin to a poor man's typing that we deem important from a methodological point of view.

```
abstract sig LeafNode extends AbstractNode {
  possibleStates : some State,
  state : possibleStates one →Time,
  possibleEventTypes : set EventType,
}
abstract sig EventType {}
abstract sig Event { // event occurrence
  instant : one Time,
  node : one LeafNode,
  type : one EventType
} { type in node.possibleEventTypes }
```

Finally, as Alloy does not feature a native notion of time, we encode it by characterizing finite traces of instants. The fact accounting for this says how states change depending on events, at every instant.

```
fact traces { // if a node state changed, there was an event concerning this node
  all t : Time, t' : t.next, n : LeafNode {
    n.state.t ≠n.state.t' implies some e : Event {
      e.instant = t
      e.node = n } } }
```

4 Fire detection example

In this section, we provide an illustration of Coy with a fire detection system in facility such as, for example, an airport or a port.

4.1 Presentation of the system

The system consists of the following components: a *smoke detector* and a *heat detector*, which are part of the automatic *fire alarm system*; a manual *fire alarm pull station*; the *local firemen*, inside the facility; and the *city firemen*, in the nearest city.

The automatic fire alarm system, which is activated by either of the two detectors, directly calls the city firemen. The manual pull station, triggered by a human present on site, calls both the local and the city firemen.

We also represent two possible failures for each of the components: (1) the loss of a component: once a component is lost, it does not send any information, (2) an erroneous failure of a component: after this kind of failure, a component sends a corrupted data (in the case of a fire detector, for instance, it can be a false alarm or a false negative). Lastly, we represent three security threats: (1) intentional wrong activation of the pull station, (2) the deactivation of the smoke detector and (3) of the heat detector.

Notice that the loss of a component and the deactivation of the smoke detector (or of the heat detector) have the same effect on a component (the availability is not ensured) although they do not have the same nature (the former is a failure, the latter is a security threat). The same applies to an erroneous failure and the intentional wrong activation of the pull station, which both affect the integrity of the information. Nevertheless, it is important to distinguish between these failure and threat events in order to allow a pure safety analysis, a pure security analysis, and a combined analysis.

4.2 Coy model

The Coy model of this system imports the Coy metamodel, declares signatures instances and relates them. Components of the system are modeled as Coy nodes. Fig 2 illustrates a particular instance of the fire detection model at a given instant. As can be seen, at this instant, all nodes are in the state OK and all ports yield a correct data (modeled by OKVal). The occurrence of an event of type failLoss (see the declaration of event types below) can be observed on the node pullStation.

Regarding the possible failures and threats mentioned above, we use the following event types, node states and possible values for the node ports.

```
one sig failLoss, failErr, threatBlock, threatPull extends EventType {}
one sig OK, Lost, Err extends State {}
one sig OKVal, LostVal, ErrVal extends Value {}
```

Then, we can declare the components and ports, as instances of the corresponding Coy concepts. For instance, here is the declaration of the fire pull station.

```
one sig pullStation extends LeafNode {} {
  input = none and output = oPullStation
  possibleStates = OK +Lost +Err and possibleEventTypes = failLoss +threatPull
}
```

The model also comprises axioms stating what happens to nodes depending on observed events. An interesting point here is that this description is declarative and does not depend on the effective nodes and ports. Concerning an event of type failLoss:

- the event can only occur on a node which is not in the state Lost,
- after the occurrence of the event, the node moves to the state Lost.

Here, we chose to model events of type threatBlock in the same way (the node also moves to the state Lost). So, they have the same effect (but they do not occur on the same components). In further analysis, if we want to distinguish between the effects of both kinds of events, we just have to use a specific node state and a specific port value corresponding to the occurrence of threatBlock.

The behavior of events of type failErr and threatPull are specified in a similar way.

```
fact behaviour {
  all e : Event | e.type in failLoss +threatBlock
    implies e.node.state.(e.instant) ≠Lost and e.node.state.(e.instant.next) = Lost
  all e : Event | e.type in failErr +threatPull
    implies e.node.state.(e.instant) = OK and e.node.state.(e.instant.next) = Err
}
```

The propagation of values is also described by an Alloy fact. For example, here is the description of the value propagation for leaf nodes with one input:

```
// leaf nodes w/ 1 input
all n : LeafNode, t : Time | {
    one n.output // tautology for this specific model, but useful if we extend it
    one n.input
  } implies {
    n.state.t = OK implies n.output.val.t = n.input.val.t
    n.state.t = Err implies n.output.val.t = ErrVal
    n.state.t = Lost implies n.output.val.t = LostVal
  }
```

4.3 Properties verification

Now we can express the safety and security properties that we want to check as Alloy *assertions*. We have mainly expressed properties related to the consequence of some failures/threats or to the robustness of the system to a given number of failures/threats. For instance, the following assertion states that whenever the smoke detector is lost (and all other nodes are OK) then the firemen can still act.

```
assert smokeDetectorLoss {
  all t : Time | {
      all n : LeafNode − smokeDetector | n.state.t = OK
      smokeDetector.state.t = Lost
    } implies (localFiremen +CityFiremen).output.val.t = OKVal
}
```

The following assertion expresses that whenever the pull station is attacked, (and all other nodes are OK) then at least one firemen department is able to act.

```
assert pullStationThreatPull{
  all t : Time | {
      all n : LeafNode − heatDetector | n.state.t = OK
      heatDetector.state.t = Err
```

Notice that the first-order quantifiers and the object-oriented syntax allows to navigate easily through the model and is convenient to state safety properties.

The following assertion expresses that whenever the smoke detector is erroneous and the pull station is attacked `threatPull`, then both local and city firemen are unable to act.

```
assert smokeDetectorFailErrPullStationThreatPull {
  all t : Time | {
      all n : LeafNode − (smokeDetector +pullStation) | n.state.t = OK
      smokeDetector.state.t = Err
      pullStation.state.t = Err
```

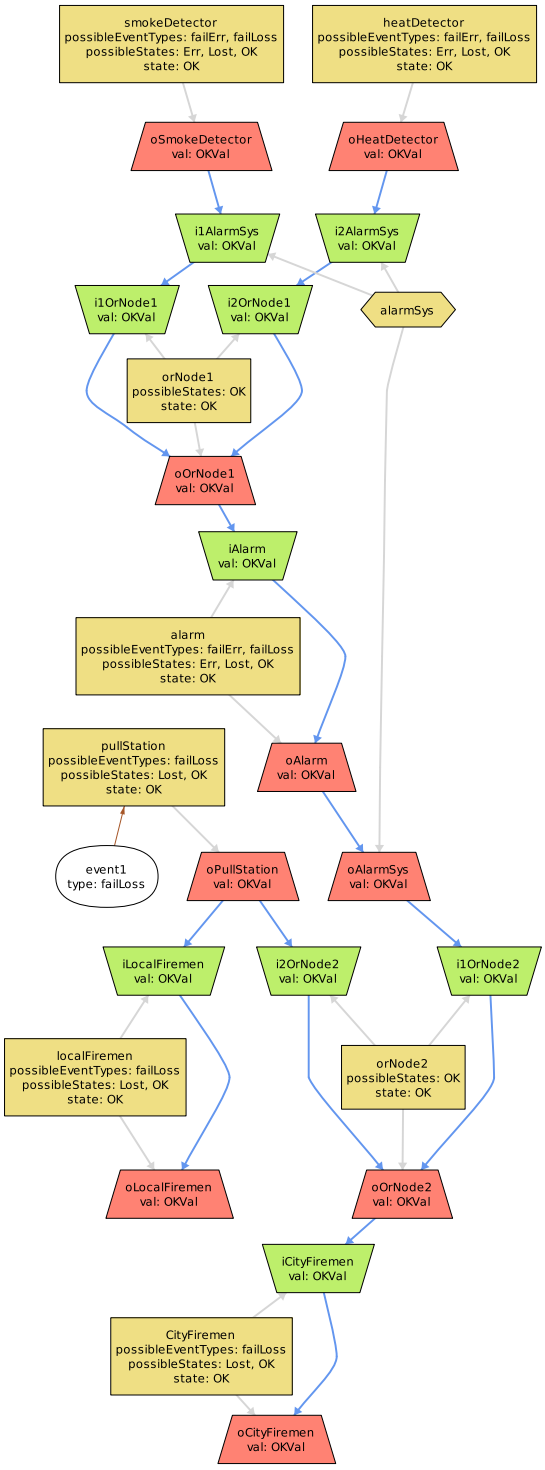


Fig. 2: Fire detection model example (see also Fig. 3 for following time steps); leaf nodes are beige rectangles, output ports are red trapeziums, input ports are green trapeziums and connections between ports are blue arrows.

The following assertions express the robustness of some parts of the system to possible failures/threats. We took benefit from the possibility to reason about a set cardinality in Alloy. Here, we count the number of events (corresponding to failures/threats) that occurred before the system enters a bad situation. For instance, the following assertion expresses that in order to make both local and city firemen unable to act properly, either the threat `threatPull` has occurred, or there has been at least two distinct failures/threats. Remark that it would have been also possible to reason independently about the number of failures and about the number of threats.

```
assert noSingleFailureThreatLeadsToFiremenNotOK {
  all t : Time | OKVal not in (localFiremen +CityFiremen).output.val.t
      implies some e : Event | e.type = threatPull and lt[e.instant, t]
```

In order to check assertions, Alloy Analyzer searches for counter-examples up to a certain bound (*i.e.* the counter-examples are such that their signatures have a cardinality less than the bound). The bound can be given by the user or chosen by the tool. In general, this bounded verification is thus incomplete: the tool may not find counter-examples whereas there are some. But in our case, the cardinality of all the signatures (`nodes`, `ports`, etc.) is fixed by the model itself. Therefore, the verification performed by Alloy Analyzer is complete.

The last four assertions have been validated by Alloy Analyzer (it does not find any counter-example).

The following assertion expresses that in order to make both local and city firemen unable to act, there has to be at least three failures/threats in the architecture.

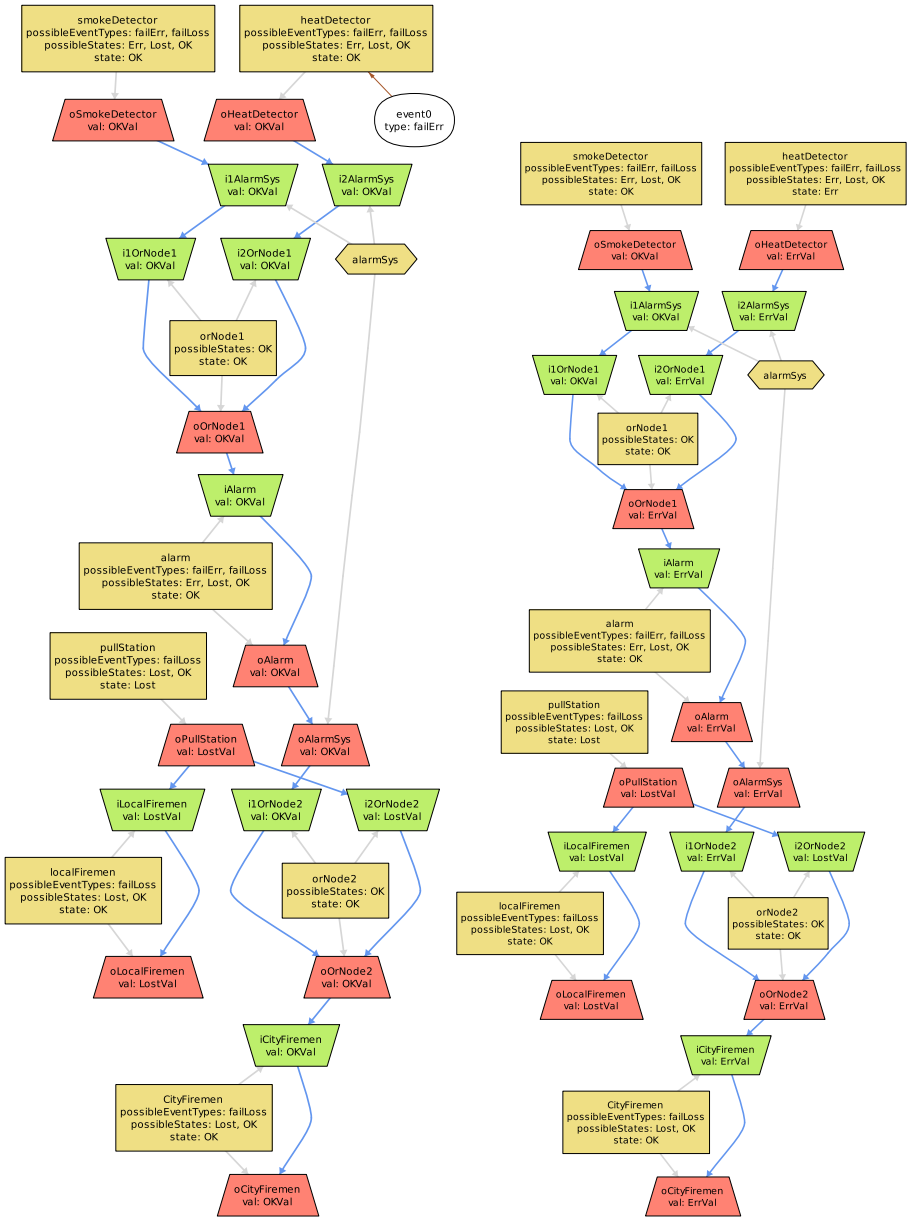
```
check noSingleFailureThreatLeadsToFiremenNotOK for 10

assert noDoubleFailureThreatLeadsToFiremenNotOK {
  all t : Time | OKVal not in (localFiremen +CityFiremen).output.val.t
```

This last assertion is not satisfied by the model. Alloy Analyzer exhibits a counter-example where the pull station and the city firemen are lost after two events (Fig. 2 shows the first time step of this counter-example, Fig 3 shows the next time steps).

5 Conclusion and Future Work

In this article, we presented a framework, called Coy, to model and assess safety and security properties of behavioral models. We have chosen to embed it in Alloy so that we can benefit from its model-based features and its expressiveness for the specification of the properties to check. A Coy model essentially describes transition systems communicating through data ports (note that other means of communication, such as synchronization of transitions, are also possible). We illustrated Coy through an example of a fire detection system, and showed it is suited to the specification of failures and threats propagation along the system. We were able to check properties over this system, and to generate counter examples for the violated properties.

In this work, the security aspects are not developed (we have just considered three threats that have the same kind of effects as failures). However, recent works have



(a) failErr event on the heat detector **(b)** final state

Fig. 3: Counter-example for assertion noDoubleFailureLeadsToFiremenNotOK (follow-up from Fig. 2)

showed the relevance of Alloy to assess more advanced security properties [10,11,6]. Moreover, it was shown that using AltaRica, one can specify the effect of richer security threats over a system architecture [2] and check related security properties. This is in favor of using Coy to model and assess security of more complex systems architectures, where both concerns are rich and interact with each other.

References

1. A. Arnold, G. Point, A. Griffault, and A. Rauzy. The altarica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40(2,3):109–124, Aug. 1999.
2. P. Bieber and J. Brunel. From safety models to security models: Preliminary lessons learnt. In A. Bondavalli, A. Ceccarelli, and F. Ortmeier, editors, *Computer Safety, Reliability, and Security. Proc. of the 1st Workshop on the Integration of Safety and Security Engineering (ISSE 2014)*, volume 8696 of *Lecture Notes in Computer Science*, pages 269–281. Springer International Publishing, 2014.
3. M. Bozzano, A. Villafiorita, O. Aakerlund, P. Bieber, C. Bougnol, E. Böde, M. Bretschneider, A. Cavallo, C. Castel, M. Cifaldi, A. Cimatti, A. Griffault, C. Kehren, B. Lawrence, A. Luedtke, S. Metge, C. Papadopoulos, R. Passarello, T. Peikenkamp, P. Persson, C. Seguin, L. Trotta, L. Valacca, and G. Zacco. Esacs: an integrated methodology for design and safety analysis of complex systems. In *Proceedings of ESREL 2003*. Balkema publisher, 2003.
4. J. Brunel, D. Chemouil, N. Mélédo, and V. Ibanez. Formal modelling and safety analysis of an avionic functional architecture with alloy. In *Embedded Real Time Software and Systems (ERTSS 2014)*, Toulouse, France, 2014.
5. J. Brunel, D. Chemouil, L. Rioux, M. Bakkali, and F. Vallée. A viewpoint-based approach for formal safety & security assessment of system architectures. In *11th Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa 2014) hosted by MODELS 2014*, volume 1235 of *CEUR-WS*, pages 39–48, 2014.
6. J. Brunel, L. Rioux, S. Paul, A. Faucogney, and F. Vallée. Formal safety and security assessment of an avionic architecture with alloy. In *Proceedings Third International Workshop on Engineering Safety and Security Systems (ESSS 2014)*, volume 150 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 8–19, 2014.
7. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499. Springer, 1999.
8. G. J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.
9. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
10. A. Lin, M. Bond, and J. Clulow. Modeling partial attacks with alloy. In B. Christianson, B. Crispo, J. Malcolm, and M. Roe, editors, *Security Protocols*, volume 5964 of *Lecture Notes in Computer Science*, pages 20–33. Springer Berlin Heidelberg, 2010.
11. M. C. Reynolds. Lightweight modeling of java virtual machine security constraints. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 146–159. Springer Berlin Heidelberg, 2010.
12. M. Toahchoodee and I. Ray. Using alloy to analyse a spatio-temporal access control model supporting delegation. *Information Security, IET*, 3(3):75–113, Sept 2009.