The role of formalism in system requirements (*additional material*)

JEAN-MICHEL BRUEL, University of Toulouse, IRIT
SOPHIE EBERSOLD, University of Toulouse, IRIT
FLORIAN GALINIER, University of Toulouse, IRIT
MANUEL MAZZARA, Innopolis University
ALEXANDR NAUMCHEV, Innopolis University
BERTRAND MEYER, Schaffhausen Institute of Technology, Innopolis University and IRIT

For space reasons, the article entitled "*The role of formalism in system requirements*" intended for the ACM Computing Surveys, by the same authors as the present one, mentions that a few of its elements are available only in an "addendum". The present document is this addendum. It contains:

- The description of a few requirements approaches listed in table 1 of the main article but not covered in detail there: NL to OCL (Natural Language to Object Constraint Language, section 4.1.A below), NL to STD (Natural Language to State Transition Diagrams, 4.1.B), URN (User Requirements Notation, 4.2.A), URML (User Requirements Modeling Language, 4.2.B), Petri Nets (4.3.A), VDM (Vienna Development Method), and Parnas's tabular relations (4.4.B).
- A supplementary bibliography covering these approaches.
- Two appendices presenting the specification of part of the main article's running example, the Landing Gear System (LGS), in two of the approaches covered in the main article: FSP/LTSA (appendix A) and Alloy (appendix B).

ACM Reference Format:
Jean-Michel Bruel, Sophie Ebersold, Florian Galinier, Manuel Mazzara, Alexandr Naumchev, and Bertrand Meyer. 2019. The role of formalism in system requirements (*additional material*). 1, 1 (November 2019), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

4. REVIEW OF IMPORTANT APPROACHES

4.1.A NL to OCL. Hähnle et al. in [16] introduced a syntax to express constrained natural language specifications. The tool is based on UML [19] and the associated OCL language [18], a Design-by-Contract-like mechanism for expressing semantic constraints on systems modeled in UML. It formalizes constraints, originally expressed in constrained natural language, into OCL (Object Constraint Language). The long-term goal is integration into the KeY Java-oriented formal verification project [2]. The presentation of NL to OCL described a tool for object-oriented modeling, intended to allow non-experts to write constraints without having to embrace full-fledged formal methods.

Authors' addresses: Jean-Michel Bruel, bruel@irit.fr, University of Toulouse, IRIT; Sophie Ebersold, sophie.ebersold@irit.fr, University of Toulouse, IRIT; Florian Galinier, florian.galinier@irit.fr, University of Toulouse, IRIT; Manuel Mazzara, m.mazzara@innopolis.ru, Innopolis University; Alexandr Naumchev, anaumchev@outlook.com, Innopolis University; Bertrand Meyer, Bertrand.Meyer@inf.ethz.ch, Schaffhausen Institute of Technology, Innopolis University and IRIT.

XXXX-XXXX/2019/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Operation retract
OCL:
```
context LGS::retract()
    pre: self.handle = up
    post: self.doors = closed and self.gears = up
```

English: for the operation retract() of the class LGS, the following precondition should hold:
the handle is up
and the following post-conditions should hold:
the doors are closed the gears are up.

Fig. 1.A Possible representation of requirement R12bis with the NL to OCL approach

Fig. 1.A expresses requirement R12bis using the NL to OCL approach. This example (not tried out since the tool is no longer available) shows how to match an English specification of the *retract* operation, including precondition and postcondition, with its representation in OCL.

Assessing NL to OCL according to the chosen criteria:

- *Scope*: the approach only covers system aspects.
- *Audience*: the target users of OCL are software engineers able to read a UML diagram and possessing a basic understanding of logic and formal reasoning.
- *Abstraction*: OCL rules are applied to UML and then Java models in this approach, covering the full spectrum from requirements to implementation.
- *Method*: the implied methodology starts from constrained natural-language requirements and expresses them in UML, with OCL for expressing semantic properties.
- *Traceability*: the requirements are expressed as contracts on operations, linking them to the UML specification. There is, however, no specific support for traceability to software artifacts.
- *Coverage*: the approach applies to requirements whose semantics can be expressed through contracts.
- *Semantics*: the OCL representation of requirements provides a semantic definition.
- *Tools*: the original tool for translation to OCL [16] is no longer available. The UML part of the approach is supported by the wide range of available UML tools (including other approaches for verification such as [8]).
- *Verifiability*: the idea is to use the KeY tool to verify consistency of requirements.

4.1.B NL to STD. The methodology of [1] iteratively transforms natural-language requirements into State Transition Diagrams, defined as sets triples [initial state, transition, resulting state].

Fig. 1.B illustrates the transformation of requirements into transition diagrams. The first step {1} transforms requirements into diagram fragments. For example, the requirement R11bis yields a transition from an unknown state, when the handle is down, to a final state *Gears locked down and doors closed*. Such states will have to be refined later. Step {2} assembles these fragments into a single diagram, representing the entire system but not final since it may still contain unknown states. Step {2} adds missing information and repeats the process.

Assessing the NL to STD approach according to the chosen criteria:

- *Scope*: the approach can cover both system and environment aspects.
- *Audience*: the requirements are first expressed in natural language, readable by all stakeholders. The transition diagrams resulting from the transformation in step 2 are meant for a more expert audience, but they include natural-language explanations coming from the original text.
- *Abstraction*: this approach is for requirements only, not influenced by implementation concerns.
- *Method*: the approach proposes a method as outlined (going from natural language to transition diagrams).

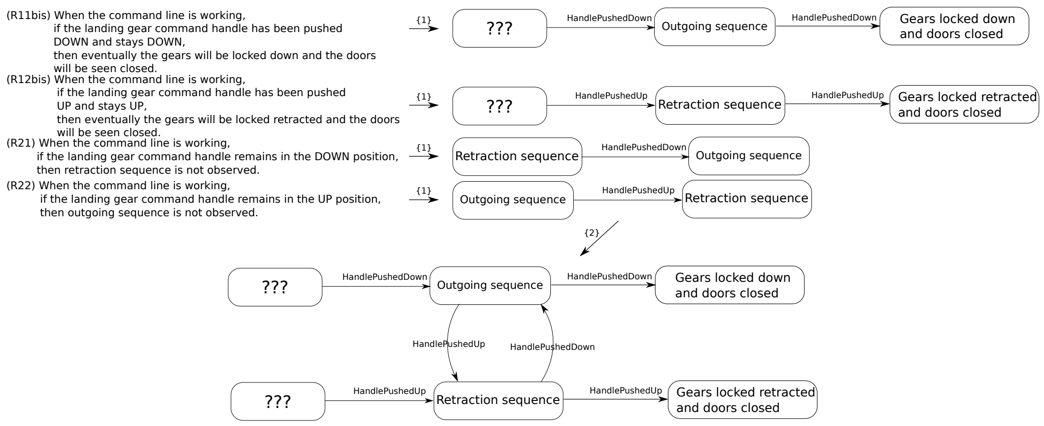
(R11bis) When the command line is working, if the landing gear command handle has been pushed DOWN and stays DOWN, then eventually the gears will be locked down and the doors will be seen closed.
(R12bis) When the command line is working, if the landing gear command handle has been pushed UP and stays UP, then eventually the gears will be locked retracted and the doors will be seen closed.
(R21) When the command line is working, if the landing gear command handle remains in the DOWN position, then retraction sequence is not observed.
(R22) When the command line is working, if the landing gear command handle remains in the UP position, then outgoing sequence is not observed.
{1}
{2}
???
HandlePushedDown
HandlePushedUp
Outgoing sequence
Retraction sequence
Gears locked down and doors closed
Gears locked retracted and doors closed

Fig. 1.B Transformation from NL requirements to STD

- *Traceability*: there is no specific support for traceability.
- *Coverage*: given that the approach expresses requirements in a very abstract form, it can include both functional and non-functional requirements.
- *Semantics*: State Transition Diagrams are a well-known notion with precise semantics.
- *Tools*: the translation into transition diagrams is manual.
- *Verifiability*: transition diagrams are formal texts, which can be submitted to tools.

4.2.A. URN. User Requirements Notation [3] is a recommendation of the International Telecommunication Union (standard Z.151, from 2008, updated 2012, third version in progress) for the modelling, analysis, specification and validation of requirements, used mainly for business process modeling.

The basic concepts are goals, scenarios, and links between such elements. URN combines two complementary views: static goals, through the Goal-oriented Requirement Language (GRL); and dynamic scenarios, through the Use Case Map (UCM) notation. Both notations support checking: for GRL models, through "strategies", representing initial situations; for UCM models, through "scenarios", similar to test cases. There is also a framework for formal verification of UCM ([11], including time extensions. For traceability:

- Internally: URN supports links connecting GRL and UCM models, enabling completeness and consistency analysis.
- With external notations: through such tools as jUCMNav [17], to integrate URN models with DOORS.

For verification, methodological elements support validating goal-oriented models and resolving conflicts ([12] [13]).

Figure 2.A uses GRL to describe the goals of the LGS example. The overall goal *LGS Safe* is decomposed into two goals (among others), reflecting R21 and R22 and refined further into more specific goals reflecting R11bis and R12bis. A task contributes to a goal: the "outgoing" task contributes to R11bis and "retracting" to R12bis. Note that tasks may depend on resources (that is not the case here).

Assessing URN according to the chosen criteria:

- *Scope*: URN is dedicated to systems and specifically to reactive systems and business systems.
- *Audience*: URN needs a specific training on GRL and UCM.

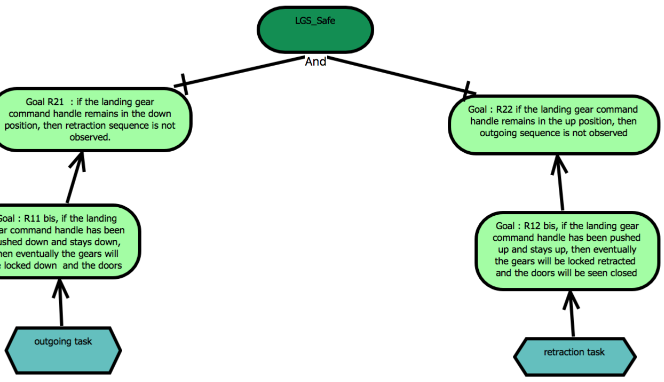
LGS_Safe
And
Goal R21 : if the landing gear command handle remains in the down position, then retraction sequence is not observed.
Goal : R22 if the landing gear command handle remains in the up position, then outgoing sequence is not observed
Goal : R11 bis, if the landing gear command handle has been pushed down and stays down, then eventually the gears will be locked down and the doors
Goal : R12 bis, if the landing gear command handle has been pushed up and stays up, then eventually the gears will be locked retracted and the doors will be seen closed
outgoing task
retraction task

Fig. 2.A Partial URN diagram for LGS requirements R11bis and R21 (*jUCMNav*)

- *Abstraction*: the approach does not have implementation concerns
- *Method*: URN does not impose any development process, but tutorials about jUCMnav present methodological elements.
- *Traceability*: as noted, URN support links between model elements.
- *Coverage*: GRL focuses on requirements, especially non-functional ones. UCM is most useful for specifying functional requirements.
- *Semantics*: from D. Amyot et al. [4], "*the URN standard describes the URN abstract and concrete syntax formally, together with well-formedness constraints. However, the semantics is currently described more informally*".
- *Tools*: jUCMNav is the open-source Eclipse [10] plugin that supports URN ([27]). Others include OpenOME [20], Sandrila [28], UniqueSoft [31], ArchSync [5] and TouchCORE [30].
- *Verifiability*: no formal support (beyond the methodological elements mentioned above).

4.2.B. URML. User Requirements Modeling Language [6, 15] is a UML profile developed in collaboration between the Technical University of Munich and Siemens. The approach unifies concepts from goal-oriented, feature-oriented, process-oriented, and risk-oriented approaches, integrating them into models covering both functional and nonfunctional requirements.

RML is a language in its own right, based on the "Meta-Object Facility" (MOF), with a meta-model mapped to a UML profile to implement the supporting tools. As a consequence, URML provides a graphical, icon-based notation to express requirements as well as associated notions such as threats, hazards, mitigations and even product lines and stakeholders.

It is possible to associate semantic properties — such as *presupposes*, *details*, *constrains*, *refines*, *with* and others — with URML links between requirements and other artifacts.

The modeling focus is on systems, but URML can also describe properties of the environment.

URML is more particularly dedicated to requirements elicitation, and does not provide any specific methodology to construct models, but it can support any methodology compatible with the URML meta-model.

Figure 2.B is an excerpt of a URML model of the LGS. To ensure a basic functionality of the LGS and its safety, we need to consider the outgoing sequence of the gears. This process is supposed to achieve the goal "when the LG command is down, gears are extended and doors are closed". To do so, it requires both R21 and R11bis. R11bis is considered a refinement of R21 since it expresses the same need at a finer level of detail.

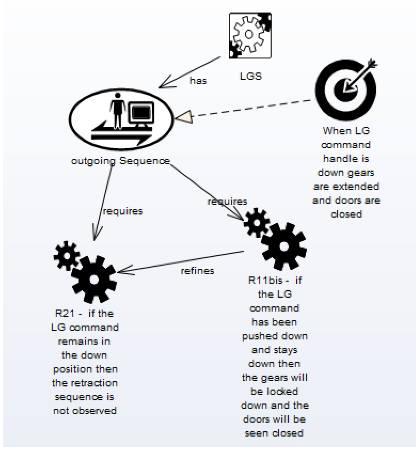
LGS
has
outgoing Sequence
When LG command handle is down gears are extended and doors are closed
requires
requires
refines
R21 - if the LG command remains in the down position then the retraction sequence is not observed
R11bis - if the LG command has been pushed down and stays down then the gears will be locked down and the doors will be seen closed

Fig. 2.B Excerpt of the URML requirements diagram of LGS (*Enterprise Architect*)

Assessing URML according to the chosen criteria:

- *Scope*: it is focused on systems and particularly complex systems.
- *Audience*: as URML is an icon-based graphical modeling language, the learning process of abstractions of this requirements language will be easy.
- *Abstraction*: URML is based on abstractions of requirements.
- *Method*: this approach does not assume a particular requirements engineering method
- *Traceability*: the links between concepts can lead to traceability.
- *Coverage*: URML integrates functional and nonfunctional requirements.
- *Semantics*: some semantics can be put on links between requirements and other artifacts.
- *Tools*: it is supported by an Enterprise Architect Add-On. [29].
- *Verifiability*: no mean is provided to do verification of requirements.

4.3.A Petri Nets. A Petri net [25] is a directed graph, where each node is either a place (circle, representing a condition) or a transition (bar) and each edge is associated with a transition. An example appears in Figure 3.A.

Petri nets have been known for several decades and have been used for many applications. Their attractiveness comes from the simplicity of the model and its ability to describe processes in a clear, visual way with a precise mathematical basis. They suffer, however, from a lack of compositionality making them unsuitable, in the view of critics, for scaling up to the description of large systems. A good analysis of the pros and cons can be found in [33].

Assessing Petri nets according to the chosen criteria:

- *Scope*: Petri nets only cover the modeling of the precise behavior of some parts of the system rather than the environment. It is hence limited to fully express requirements of a system.
- *Audience*: the notation, while graphical and clear, has to be learned.
- *Abstraction*: this approach is for requirements only, not influenced by implementation concerns.
- *Method*: the approach does not assume a particular requirements engineering method.
- *Traceability*: the approach focuses on the expression of dynamic behavior requirements, and offers no specific support for traceability.
- *Coverage*: Petri nets in their basic form address functional requirements.

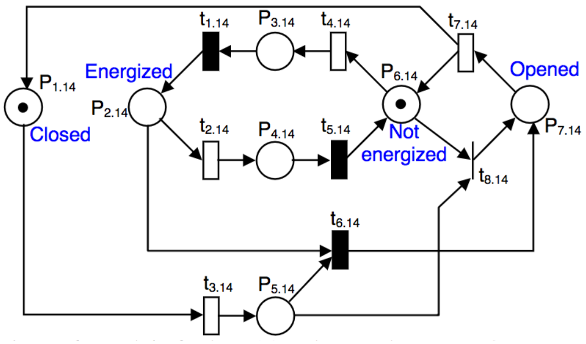

Fig. 3.A Petri Net model of an Electrical Circuit of Negative Pressuring Electro-valve (taken from [32])

- *Semantics*: Petri nets have an exact mathematical definition of their execution semantics, with a well-developed mathematical theory for process analysis.
- *Tools*: Petri nets have been around for a long time; many tools have been developed to edit, execute and verify nets.
- *Verifiability* is supported by Petri-net tools, relying on model-checking.

4.4.A Vienna Development Method. The Vienna Development Method (VDM) [7], originally developed at the IBM Laboratory in Vienna in the 1970s, was one of the first formal methods in the history of software and system modeling. It includes the VDM Specification Language (VDM-SL) [26] and its extended form (VDM++) [9]. VDM-SL uses modules, while VDM++ applies object-oriented structuring with classes and multiple inheritance. Computing systems may be modeled in VDM-SL at a high level of abstraction, then transformed into progressively more detailed designs through a refinement process (*reification*) similar to the Event-B ideas.

VDM-SL allows users to express different states of the systems and invariants that shall be met. As an example, Figure 4.A expresses requirements R21 and R22.

```
state LGS of
  gears: <retracted> | <extended> | <retracting> | <extending>
  doors: <opened> | <closed> | <opening> | <closing>
  handle: <down> | <up>
  inv mk_LGS(gears, doors, handle) ==
    (handle =<down> => gears <> <retracted>) and
    (handle =<up> => gears <> <extended>)
  init lgs ==lgs =mk_LGS(<extended>,<closed>,<down>)
end
```

Figure 4.A. VDM-SL representation of the LGS states, with requirements R21 and R22

Another way to introduce requirements into a VDM specification is to translate the requirements into pre- and post-conditions of operations. For example, the requirement R11bis (resp. R12bis) is a description of what happens when handle remains down (up). Figure 4.B specifies these operations; the **pre** clause expresses the condition under which the operation may be called, and the **post** clause characterizes the state after execution.

```
operations
```

```
  extension_sequence()
   ext wr gears
       wr doors
       rd handle
   pre handle =<down>
   post handle =<down> => (gears =<extended> and doors =<closed>);

  retraction_sequence()
   ext wr gears
       wr doors
       rd handle
   pre handle =<up>
   post handle =<up> => (gears =<retracted> and doors =<closed>);
end LGS
```

Figure 4.B VDM-SL operations specified by requirements R11bis and R12bis

The **rd** (read-only) and **wr** (read-write) clauses specify which parts of the LGS the operations may access and modify.

Assessing VDM according to the chosen criteria:

- *Scope*: The approach covers the expression of requirements on both the system and environment aspects.
- *Audience*: VDM requires an understanding of mathematical foundations.
- *Abstraction*: VDM is a mathematical formalism, suitable for the early stages of the development process.
- *Method*: VDM supports the description of data and functionality. Data are defined by means of types. Functionality is defined in terms of operations.
- *Traceability*: VDM has no specific support for traceability.
- *Coverage*: VDM covers the functional part of requirements.
- *Semantics*: VDM has a formal semantics, published as an ISO (International Standards Organization) standard.
- *Tools*: VDM is mature and has an extensive tool support (e.g., Overture[21]).
- *Verifiability*: Supporting tools are available to verify VDM specifications.

4.4.B. Tabular relations. A general approach to requirements introduced by David Parnas [22] relies on specifying relations between system variables, represented in tabular form. The relations can be n-dimensional for any n. The underlying mathematical theory, relations over a collection of sets, is well known, and also serves as a basis for the relational model of databases.

Figures 4.C and 4.D illustrate an application of the tabular approach to the LGS system. To capture requirements R11bis and R12bis, since tabular representations do not support temporal operators, we introduce variables $gear\_status'$ and $door\_status'$, denoting the values of $gear\_status$ and $door\_status$ when the program finishes its execution.

The header of the table in Fig. 4.C enumerates the possible conditions, where $NC(handle\_position)$ reflects the assumption that the position of the handle does not change [23]: $handle\_position = down \wedge NC(handle\_position)$, and $handle\_position = up \wedge NC(handle\_position)$. Requirements R_21 and R_22 talk about an immediate, rather than temporal, response of the system to the stimulus coming from the handle.

The resulting vector relation table in Fig. 4.D thus does not contain any additional variables. An important property of a table is to be *proper* — that is, to characterize mutually disjoint situations.

	$handle\_position = down \wedge NC(handle\_position)$	$handle\_position = up \wedge NC(handle\_position)$
$gear\_status'$	$extended$	$retracted$
$door\_status'$	$closed$	$closed$

Fig. 4.C Expressing requirements R11bis and R12bis with a vector function table [22]. The $NC(handle\_position)$ predicate states that the value of variable $handle\_position$ does not change [23].

The LGS knows two situations: when the pilots' handle in the cockpit stays (1) down and (2) up. The LGS environment guarantees mutual disjointness of these situations.

	$handle\_position = down$	$handle\_position = up$
$gear\_status$	$gear\_status \neq retracting$	$gear\_status \neq extending$

Fig. 4.D Expressing requirements R21 and R22 with a vector relation table [22]. The table describes the relation connecting the handle's position and the gear's status.

Assessing the tabular relations approach according to the chosen criteria:

- *Scope*: the approach can cover all aspects of requirements, including both system and environment aspects.
- *Audience*: the tabular notation is formal and requires mathematical qualification.
- *Abstraction*: the notation operates at the level of program variables. Tables specify programs through specifying how the programs' executions affect the variables' values under mutually disjoint preconditions.
- *Method*: the approach does not assume or promote a particular requirements engineering method.
- *Traceability*: since tables directly use program variables, traceability to implementation would fit naturally, assuming appropriate tool support.
- *Coverage*: the tabular notation focuses on specification of functional requirements. It defines ten classes of tables for this purpose [22].
- *Semantics*: the semantics of tabular relations is formally defined [22], [23].
- *Tools*: while the approach, with its precise definition and simple tabular representation, would naturally lend itself to tool support, we found no record of such tools in the published literature.
- *Verifiability*: verification techniques that work with tabular relations include automated consistency checking [14] and test oracle generation [24].

REFERENCES

[1] D. Aceituna, H. Do, G.S. Walia, and S.-W. Lee. 2011. Evaluating the use of model-based requirements verification method: A feasibility study. In *Workshop on Empirical Requirements Engineering (EmpiRE 2011)*. IEEE, 13–20.

[2] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. 2005. The Key Tool. *Software & Systems Modeling* 4, 1 (2005), 32–54.

[3] D. Amyot. 2003. Introduction to the User Requirements Notation: learning by example. *Comp. Networks* 42, 3 (June 2003), 285–301.

[4] Daniel Amyot and Gunter Mussbacher. 2011. User Requirements Notation: The First Ten Years, The Next Ten Years (Invited Paper). *J. of Software* 6, 5 (2011), 747–768.

[5] ArchSync. 2006. (2006). https://sourceforge.net/projects/archsync/

[6] Brian Berenbach, Florian Schneider, and Helmut Naughton. 2012. The use of a requirements modeling language for industrial applications. In *2012 20th IEEE Springer Requirements Engineering Conf. (RE)*. IEEE, 285–290.

[7] D. Bjørner and C. B. Jones (Eds.). 1978. *Vienna Development Method: The Meta-Language*. LNCS, Vol. 61. Springer.

[8] Jordi Cabot, Robert Clarisó, and Daniel Riera. 2007. UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming. In *Proc. Twenty-second IEEE/ACM Int. Conf. on Automated Software Engineering (ASE '07)*. ACM, 547–548.

[9] Eugene Durr and Jan van Katwijk. 1992. VDM++, a formal specification language for object-oriented designs. In *CompEuro 1992 Proceedings Computer Systems and Software Engineering*. IEEE Computer Society Press, 214–219.

[10] Eclipse foundation. 2004. Eclipse Foundation. (2004). http://www.eclipse.org

[11] Jameleddine Hassine. 2010. AsmL-Based Concurrency Semantic Variations for Timed Use Case Maps. In *Int. Conf. on Abstract State Machines, Alloy, B and Z*, Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves (Eds.). Springer, 34–46.

[12] Jameleddine Hassine and Daniel Amyot. 2016. A Questionnaire-based Survey Methodology for Systematically Validating Goal-oriented Models. *Requir. Eng.* 21, 2 (jun 2016), 285–308.

[13] Jameleddine Hassine and Daniel Amyot. 2017. An Empirical Approach Toward the Resolution of Conflicts in Goal-oriented Models. *Softw. Syst. Model.* 16, 1 (Feb. 2017), 279–306.

[14] Constance L Heitmeyer, Ralph D Jeffords, and Bruce G Labaw. 1996. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 3 (1996), 231–261.

[15] J. Helming, M. Koegel, F. Schneider, M. Haeger, C. Kaminski, B. Bruegge, and B. Berenbach. 2010. Towards a unified Requirements Modeling Language. In *2010 5th Int. Workshop on Requirements Engineering Visualization*. IEEE, 53–57.

[16] R. Hähnle, K. Johannisson, and A. Ranta. 2002. An Authoring Tool for Informal and Formal Requirements Specifications. In *Fund. Approaches to Software Engineering, LNCS 2306*, R.-D. Kutsche and H. Weber (Eds.). Springer, 233–248.

[17] Jucmnavc. 2017. (2017). https://www.openhub.net/p/jucmnav

[18] Object Management Group (OMG). 2014. OCL 2.4. (Feb. 2014). http://www.omg.org/spec/OCL/2.4/ http://www.omg.org/spec/OCL/2.4/.

[19] Object Management Group (OMG). 2015. UML 2.5. (March 2015). http://www.omg.org/spec/UML/2.5/

[20] OpenOme. 2000. (2000). https://se.cs.toronto.edu/trac/ome

[21] Overture. 2020. (2020). http://overturetool.org/method/

[22] David L. Parnas. 1992. *Tabular representation of relations*. Communications Research Laboratory Report, McMaster University.

[23] David L. Parnas. 1993. Predicate logic for software engineering. *IEEE Transactions on Software Engineering* 19, 9 (1993), 856–862.

[24] Dennis K. Peters and David L. Parnas. 1998. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering* 24, 3 (1998), 161–173.

[25] James L. Peterson. 1977. Petri Nets. *ACM Comput. Surv.* 9, 3 (Sept. 1977), 223–252.

[26] The Overture Project. 2017. Overture Tool: Formal Modelling in VDM. (2017). http://overturetool.org/method/

[27] Jean-François Roy, Jason Kealey, and Daniel Amyot. 2006. Towards Integrated Tool Support for the User Requirements Notation. In *System Analysis and Modeling: Language Profiles: 5th Int. Workshop, SAM 2006, Kaiserslautern, Germany, May 31 - June 2, 2006, Revised Selected Papers*, Reinhard Gotzhein and Rick Reed (Eds.). Springer, 198–215. DOI: 10.1007/11951148_13.

[28] Sandrila. 2019. (2019). http://www.paulherber.co.uk/visio-sdl/index.php

[29] Sparx Systems. 2017. Enterprise Architect. (2017). https://sparxsystems.eu/enterprisearchitect/system-requirements/

[30] TouchCORE. 2020. (2020). http://touchcore.cs.mcgill.ca

[31] UniqueSoft. 2020. (2020). http://www.uniquesoft.com

[32] Emilia Villani, Fabrício Junqueira, Paulo Eigi Miyagi, and Robert Valette. 2004. Petri net and OO for the modular analysis of an aircraft landing system. *ABCM Symp. Series in Mechatronics* 1 (11 2004), 570–579. http://www.abcm.org.br/symposium-series/SSM_Vol1/Section_IV_Discrete_Event_Dynamic_Systems/SSM_IV_08.pdf

[33] Richard Zurawski and MengChu Zhou. 1994. Petri nets and industrial applications: A tutorial. *IEEE Trans. Industrial Electronics* 41, 6 (1994), 567–583.

A LGS IN FSP/LTSA

```
// Reaction of the LGS to the different states of the handle.
LGS_BEHAVIOR = (do_nothing -> LGS_BEHAVIOR | up -> OPEN_FOR_RETRACTION | down -> OPEN_FOR_EXTENSION),
OPEN_FOR_RETRACTION = (open -> START_RETRACTION),
START_RETRACTION = (start_retraction -> END_RETRACTION),
END_RETRACTION = (end_retraction -> START_CLOSING_RETRACTED | down -> START_EXTENSION),
START_CLOSING_RETRACTED = (start_closing -> END_CLOSING_RETRACTED),
END_CLOSING_RETRACTED = (end_closing -> LGS_BEHAVIOR | down -> OPEN_FOR_EXTENSION),
OPEN_FOR_EXTENSION = (open -> START_EXTENSION),
START_EXTENSION = (start_extension -> END_EXTENSION),
END_EXTENSION = (end_extension -> START_CLOSING_EXTENDED | up -> START_RETRACTION),
START_CLOSING_EXTENDED = (start_closing -> END_CLOSING_EXTENDED),
END_CLOSING_EXTENDED = (end_closing -> LGS_BEHAVIOR | up -> OPEN_FOR_RETRACTION).
// Fluents that model possible states of the LGS:
fluent HANDLE_IS_DOWN = <{down}, {up}>
fluent HANDLE_IS_UP = <{up}, {down}>
fluent DOOR_IS_CLOSING = <{start_closing}, {end_closing, open}>
fluent DOOR_IS_CLOSED = <{end_closing}, {open}>
fluent GEAR_IS_EXTENDING = <{start_extension}, {end_extension, start_retraction}>
fluent GEAR_IS_EXTENDED = <{end_extension}, {start_retraction}>
fluent GEAR_IS_RETRACTING = <{start_retraction}, {end_retraction, start_extension}>
fluent GEAR_IS_RETRACTED = <{end_retraction}, {start_extension}>
// Specifying the LGS control handle.
CONTROL_HANDLE = (down -> INITIALLY_DOWN | up -> INITIALLY_UP),
INITIALLY_DOWN = (up -> down -> INITIALLY_DOWN),
INITIALLY_UP = (down -> up -> INITIALLY_UP).
// Modelling different modes of operation.
||LGS = (LGS_BEHAVIOR || CONTROL_HANDLE) >>
{up}. // Uncomment to model LGS with the handle pushed down.
// {down}. // Uncomment to model LGS with the handle pulled up.
assert EVENTUALLY_ALWAYS_DOWN = <> [] HANDLE_IS_DOWN
assert R11bis = [] ([] HANDLE_IS_DOWN -> <> [] (GEAR_IS_EXTENDED && DOOR_IS_CLOSED))
assert R21 = [] ([] HANDLE_IS_DOWN -> [] ! GEAR_IS_RETRACTING)
assert EVENTUALLY_ALWAYS_UP = <> [] HANDLE_IS_UP
assert R12bis = [] ([] HANDLE_IS_UP -> <> [] (GEAR_IS_RETRACTED && DOOR_IS_CLOSED))
assert R22 = [] ([] HANDLE_IS_UP -> [] ! GEAR_IS_EXTENDING)
```

B LGS IN ALLOY

```
abstract sig Handle {}
sig Down, Up extends Handle {}{Down + Up =Handle}
abstract sig Door {}
sig Closed, Opening, Open, Closing extends Door {}{
  Closed + Opening + Open + Closing =Door
}
abstract sig Gear {}
sig Extended, Retracting, Retracted, Extending extends Gear {}{
  Extended + Retracting + Retracted + Extending =Gear
}
sig LGS {
  handle: Handle,
  door: Door,
  gear: Gear
} {
  (gear in Retracting ∨gear in Extending) implies (door in Open)
  (handle in Up) implies (gear not in Extending)
}
pred closeDoor [lgs, lgs' :LGS] {
  (lgs.door in Open) implies (lgs'.door in Closing)
  (lgs.door in Closing) implies (lgs'.door in Closed)
  (lgs.door in Opening) implies (lgs'.door in Closing)
  (lgs.door in Closed) implies (lgs'.door in Closed)
  (lgs.gear =lgs'.gear)
}
pred openDoor [lgs, lgs' :LGS] {
  (lgs.door in Open) implies (lgs'.door in Open)
  (lgs.door in Closing) implies (lgs'.door in Opening)
  (lgs.door in Opening) implies (lgs'.door in Open)
  (lgs.door in Closed) implies (lgs'.door in Opening)
  (lgs.gear =lgs'.gear)
}
pred retractGear [lgs, lgs' :LGS] {
  (lgs.gear in Extended) implies (lgs'.gear in Retracting)
  (lgs.gear in Retracting) implies (lgs'.gear in Retracted)
  (lgs.gear in Extending) implies (lgs'.gear in Retracting)
  (lgs.gear in Retracted) implies (lgs'.gear in Retracted)
  (lgs.door =lgs'.door)
}
```

```
pred extendGear [lgs, lgs' :LGS] {
  (lgs.gear in Extended) implies (lgs'.gear in Extended)
  (lgs.gear in Retracting) implies (lgs'.gear in Extending)
  (lgs.gear in Extending) implies (lgs'.gear in Extended)
  (lgs.gear in Retracted) implies (lgs'.gear in Extending)
  (lgs.door =lgs'.door)
}
pred retractionSequence [lgs, lgs' :LGS] {
  (lgs.gear not in Retracted) implies openDoor [lgs, lgs']
  ((lgs.gear not in Retracted) ∧(lgs.door in Open)) implies retractGear [lgs, lgs']
  (lgs.gear in Retracted) implies closeDoor [lgs, lgs']
}
pred outgoingSequence [lgs, lgs' :LGS] {
  (lgs.gear not in Extended) implies openDoor [lgs, lgs']
  ((lgs.gear not in Extended) ∧(lgs.door in Open)) implies extendGear [lgs, lgs']
  (lgs.gear in Extended) implies closeDoor [lgs, lgs']
}
pred Main [lgs, lgs' :LGS] {
  (lgs.handle in Up) implies retractionSequence [lgs, lgs']
  (lgs.handle in Down) implies outgoingSequence [lgs, lgs']
}
run Main for exactly 2 LGS, 2 Handle, 4 Door, 4 Gear
R21: check {
 all lgs, lgs': LGS |((lgs.handle in Down) and (lgs'.handle in Down) and Main [lgs, lgs'])
      ↪ implies (lgs'.gear not in Retracting)
} for 5
R22: check {
 all lgs, lgs': LGS |((lgs.handle in Up) and (lgs'.handle in Up) and Main [lgs, lgs'])
      ↪ implies (lgs'.gear not in Extending)
} for 5
R11bis: check {
 all lgs1, lgs2, lgs3, lgs4 :LGS |
   (((lgs1.handle in Down and lgs2.handle in Down and lgs3.handle in Down and lgs4.handle in
        ↪ Down) and
   (Main [lgs1, lgs2] and Main [lgs2, lgs3] and Main [lgs3, lgs4]))
    implies (lgs4.gear in Extended and lgs4.door in Closed))
} for 5
R12bis: check {
 all lgs1, lgs2, lgs3, lgs4 :LGS |
   (((lgs1.handle in Up and lgs2.handle in Up and lgs3.handle in Up and lgs4.handle in Up) and
   (Main [lgs1, lgs2] and Main [lgs2, lgs3] and Main [lgs3, lgs4]))
    implies (lgs4.gear in Retracted and lgs4.door in Closed))
} for 5
```