From boilerplated requirements to Alloy: half-way between text and formal model

Daniel Cadete, Alcino Cunha, José M. Faria,
José N. Oliveira and André Passos

Techn. Report TR-HASLab:04:2012

Jul. 2012

HASLab - High-Assurance Software Laboratory
Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://haslab.di.uminho.pt

TR-HASLab:04:2012

From boilerplated requirements to Alloy: half-way between text and formal model

by Daniel Cadete, Alcino Cunha, José M. Faria,
José N. Oliveira and André Passos

Abstract

Getting system requirements right is still one of the highest challenges in critical systems development. Having different kinds of stakeholders involved demands for a common language of communication, which hinders the use of formal languages.

This document presents an approach that links natural language requirements and formal languages through boilerplates — a set of predefined templates with gaps to fill. The methodology is supported by PROVA, a tool that automatically translates the (boilerplated) requirements into Alloy, enabling early detection of ambiguities and inconsistencies through model checking.

From boilerplated requirements to Alloy: half-way between text and formal model

Daniel Cadete, Alcino Cunha, José M. Faria, José N. Oliveira and André Passos

Jul. 2012

Abstract

Getting system requirements right is still one of the highest challenges in critical systems development. Having different kinds of stakeholders involved demands for a common language of communication, which hinders the use of formal languages.

This document presents an approach that links natural language requirements and formal languages through boilerplates — a set of predefined templates with gaps to fill. The methodology is supported by PROVA, a tool that automatically translates the (boilerplated) requirements into Alloy, enabling early detection of ambiguities and inconsistencies through model checking.

From boilerplated requirements to Alloy: half-way between text and formal model

Daniel Cadete[1], Alcino Cunha[1], José M. Faria[2], José N. Oliveira[1], and André Passos[2]

[1] High Assurance Software Lab / INESC TEC, Univ. Minho
{dcadete,alcino,jno}@di.uminho.pt haslab.di.uminho.pt
[2] Educed Lda, Portugal
{jmf,abp}@educed-emb.com www.educed-emb.com

Abstract. Getting system requirements right is still one of the highest challenges in critical systems development. Having different kinds of stakeholders involved demands for a common language of communication, which hinders the use of formal languages. This paper presents an approach that links natural language requirements and formal languages through boilerplates – a set of predefined templates with gaps to fill. The methodology is supported by PROVA, a tool that automatically translates the (boilerplated) requirements into Alloy, enabling early detection of ambiguities and inconsistencies through model checking.

1 Introduction

Behavior failures in mission- or safety-critical systems may have very severe consequences. Thus developing and verifying these kinds of system poses significant challenges to engineers. Considerable effort has been put into software verification and source code testing. Yet, contrary to the general public perception, most software failures are not due to bugs introduced in the coding stage: by far, the largest class of serious software problems can be traced to errors made in the eliciting, specification, and analysis of *requirements* [4].

A key source for mistakes arises from the fact that requirements Instead, they are written by system experts and exchanged among the different stakeholders of the project in natural language. Quite easily, requirements texts are not clear and developers misinterpret them.

Much research has been carried out to develop techniques that trim down ambiguity and lack of precision in requirements documents. Notable examples include works in the automatic evaluation of the quality of the requirements text [8], in the identification of patterns leading to ambiguity [5], procedures for rewriting requirements [7], and definition of controlled languages for requirements specification [6]. These approaches are valuable for improving the way requirements are written and communicated between stakeholders. They miss, however, to evaluate the correctness of the specified requirements.[3]

[3] The work in controlled natural languages is admittedly too broad to survey in a few words; for an excellent starting reference please refer to `http://sites.google.com/`

The introduction of formal technologies in the global process can provide rigorous and machinery support for correctness verification, identifying inconsistencies, ambiguities, and omissions. Their adoption in industry faces, however, a number of difficulties. Above all and most referred, the average software analyst is illiterate in formal methods[4]. This paper proposes a technique for formalizing requirements while hiding this complexity from the user, who keeps writing the requirements in natural language form. The idea is to combine the use of so-called *boilerplated* text [2] with formal modeling, enabling early detection of ambiguities and inconsistencies through model checking. The methodology is supported by PROVA, a tool that automatically translates the (boilerplated) requirements into Alloy [3].

2 Approach: Boilerplated requirements

Deriving computer programs from textual requirements remains a challenging and error-prone activity despite the many attempts to (semi)automate the process. The gap between what the user requires and the final code

$$Requirements \xrightarrow{\qquad\qquad\qquad\qquad} Code$$

simply is too wide, creating a space for misinterpretation. It is generally accepted that such a gap should be split into two shorter paths by putting an *abstract model* of the system to be designed in between:

$$Requirements \longrightarrow Abstract\ model \longrightarrow Code$$

Such an intermediate step can be achieved with variable degree of formality, be this a mathematical model, a UML collection of diagrams, or other. Many will find the first step still too hard, calling again for something in between. *Boilerplated* text [2] can be of help in this respect, leading to:

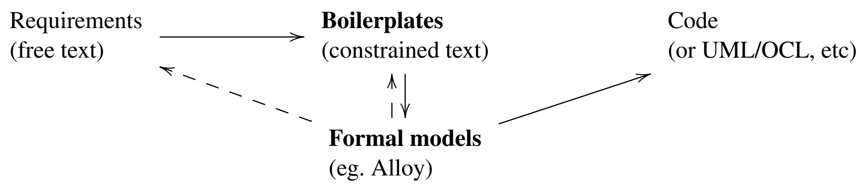
$$Requirements \longrightarrow \textbf{Boilerplates} \longrightarrow Abstract\ model \longrightarrow Code \qquad (1)$$

In a sense, a boilerplate is nothing but parametric text (i.e. text with *placeholders* to be filled in) bridging the gap between unconstrained and semi-formal expression. While reducing ambiguity, it offers a stable ground for automatic translation to formal notation, while keeping the natural language appearance.

Building a suitable and comprehensive boilerplate repository remains a challenge. The rationale in [2] is to find (statistically) frequent patterns in textual requirements. This leaves, however, still open the problem of generating models or code from such a framework. The alternative we propose relies on finding such patterns not in textual data but rather in *formal models*:

`site/controllednaturallanguage`, or the annual conference in the field, `http://attempto.ifi.uzh.ch/site/cnl2012`.

[4] Even when tools are available alleviating the burden of understanding a formal method, very often the formulæ which encode the problem are too complex, as well explained in [1].



We identify repetitive formal specification patterns in modeling languages and derive boilerplates from them. Clearly, this method calls for expertise in formal methods, but once a boilerplate is enrolled in the repository one knows that (a) it corresponds to an abstract model proved useful, and (b) the derivation of such a model from instances of the boilerplate is ensured by construction.

In the current paper we adopt Alloy [3] as our formal language and main inspiration for boilerplate construction. Its simplicity reduces pattern inspection a great deal and captures both ontological/structural aspects and dynamic requirements. Alloy's *navigation style* emerging from its "dot join" notation is much closer to natural language than quantifier-full sentences in first order logic. Alloy also enables *model checking*, a way to grant early feedback to the user.

3 Boilerplates Repository

The proposed boilerplates can either be structural, declaring the entities in the domain and relationships between them, or behavioral, specifying how the declared relationships should behave over time. Currently, structural boilerplates follow the grammar:

$$\begin{array}{rl}
\mathit{structural} ::= & \texttt{every}\ \mathit{entity}\ \texttt{shall have}\ [\mathit{mult}]\ [\texttt{fixed}]\ [\mathit{attribute}]\ \mathit{entity} \\
& \mid \mathit{quantifier\ entity}\ \texttt{shall contain}\ [\mathit{mult}]\ [\texttt{fixed}]\ [\mathit{attribute}]\ \mathit{entity} \\
& \mid \mathit{quantifier\ entity}\ \texttt{is a}\ \mathit{entity} \\
& \mid \mathit{quantifier\ entity}\ \texttt{shall be able to}\ \mathit{action\ entity} \\
\mathit{mult} ::= & \texttt{one} \mid \texttt{at most one} \mid \texttt{some} \\
\mathit{entity} ::= & \mathit{noun},\ \mathit{attribute} ::= \mathit{adjective} \mid \mathit{noun},\ \mathit{action} ::= \mathit{verb}
\end{array}$$

The following relationships are taken into account: *association* (`shall have`); *composition* (`shall contain`); *generalization* (`is a`). Associations and compositions can optionally be declared as immutable (`fixed`), and be given an explicit attribute name and multiplicity. We also allow the specification of a *capability* (`shall be able to`), meaning that an entity can somehow act upon another entity. Structural boilerplates are translated to Alloy as follows:

$$\begin{array}{rcl}
[\![\texttt{every}\ e_1\ \texttt{shall have}\ m\ \texttt{fixed}\ a\ e_2]\!] & \equiv & \texttt{sig}\ e_1\ \{\ a : [\![m]\!]\ e_2\ \} \\
& & \texttt{sig}\ e_2\ \{\ \} \\
[\![\texttt{every}\ e_1\ \texttt{shall have}\ m\ a\ e_2]\!] & \equiv & \texttt{sig}\ e_1\ \{\ a : e_2\ [\![m]\!] \rightarrow \texttt{Time}\} \\
& & \texttt{sig}\ e_2\ \{\ \} \\
[\![\texttt{every}\ e_1\ \texttt{shall contain}\ m\ \texttt{fixed}\ a\ e_2]\!] & \equiv & \texttt{sig}\ e_1\ \{\ a : [\![m]\!]\ e_2\ \} \\
& & \texttt{sig}\ e_2\ \{\ e_1 : \texttt{lone}\ e_1\ \} \\
& & \texttt{fact}\ \{\ e_1 = \tilde{}e_2\ \}
\end{array}$$

$$\begin{aligned}
[\![\texttt{every}\ e_1\ \texttt{shall contain}\ m\ a\ e_2]\!] &\equiv \texttt{sig}\ e_1\ \{\ a : e_2\ [\![m]\!] \rightarrow \texttt{Time}\} \\
&\ \texttt{sig}\ e_2\ \{\ e_1 : e_1\ \texttt{lone} \rightarrow \texttt{Time}\} \\
&\ \texttt{fact}\ \{\ \texttt{all t : Time} \mid e_1.\texttt{t} = \tilde{}(a.\texttt{t})\ \} \\
[\![q\ e_1\ \texttt{is a}\ e_2]\!] &\equiv \texttt{sig}\ e_1\ \texttt{extends}\ e_2\ \{\ \} \\
&\ \texttt{sig}\ e_2\ \{\ \} \\
[\![\texttt{every}\ e_1\ \texttt{shall be able to}\ a\ e_2]\!] &\equiv \texttt{sig}\ e_1\ \{\ a : e_2\ \texttt{lone} \rightarrow \texttt{Time}\} \\
&\ \texttt{sig}\ e_2\ \{\ \}
\end{aligned}$$

Translation of multiplicities is trivial and is omitted. We only present the translation of boilerplates with an explicit attribute name. If not present, the name of the targeted entity is used as default. The rules only describe how each boilerplate is translated in isolation. The results of translating different boilerplates are then combined with a *coalesced sum* operator $\oplus$ that, among other things, merges different declarations for the same signature, as exemplified below:

$$\begin{array}{l} \texttt{sig A \{ r : B \}} \\ \texttt{sig B \{ \}} \end{array} \oplus \begin{array}{l} \texttt{sig A \{ s : C \}} \\ \texttt{sig C \{ \}} \end{array} = \begin{array}{l} \texttt{sig A \{ r : B, s : C \}} \\ \texttt{sig B, C \{ \}} \end{array}$$

$$\begin{array}{l} \texttt{sig A \{ r : B \}} \\ \texttt{sig B \{ \}} \end{array} \oplus \begin{array}{l} \texttt{sig A \{ r : C \}} \\ \texttt{sig C \{ \}} \end{array} = \begin{array}{l} \texttt{sig A \{ r : B + C \}} \\ \texttt{sig B, C \{ \}} \end{array}$$

One of the hallmarks of Alloy is it simplicity. In particular, there is no predefined syntax to model dynamic behavior, and the atoms that populate model instances are all immutable. A typical workaround to this limitation is to introduce a special signature to identify global system states (or different points in time). Then, to specify a mutable relation an extra state (or time) column is added to its signature. Following this strategy, we will introduce a special signature `Time` in the generated model. To model traces, we will use a predefined Alloy model to impose a total order on `Time` atoms.

To exemplify in more detail, the translation of a composition boilerplate is as follows: besides the relation between a component and its part, we also introduce a contained relation from the part to its component with the name of the former, allowing us to unambiguously mention it in the behavioral requirements. This relation must be simple, i.e. have multiplicity `lone`, since a part cannot be shared between components, and should be symmetric in relation to the contains relation. If the composition is mutable, both these relations are extended with `Time` and the aforementioned constraints become invariants over execution traces. As an example of composition consider the following requirement on channels:

$$\begin{aligned}
&[\![\mathit{Every\ Channel}\ \texttt{shall contain}\ \mathit{Message}\texttt{s}]\!] \equiv \\
&\texttt{sig}\ \mathit{Channel}\ \{\ \mathit{message} : \mathit{Message}\ \texttt{set} \rightarrow \texttt{Time}\ \} \\
&\texttt{sig}\ \mathit{Message}\ \{\ \mathit{channel} : \mathit{Message}\ \texttt{lone} \rightarrow \texttt{Time}\ \} \\
&\texttt{fact}\ \{\ \texttt{all t : Time} \mid \mathit{channel}.\texttt{t} = \tilde{}(\mathit{message}.\texttt{t})\ \}
\end{aligned}$$

The behavioral boilerplates are inspired by the popular navigational style of Alloy, where the most common constraints enforce the cardinality or inclusion of sets computed by navigating using the relational composition operator. This navigational style is elegantly captured in English by the possessive form, leading to the boilerplates described in the following grammar:

$$\begin{aligned}
\mathit{behavioral} &::= \mathit{lset}\ \texttt{shall}\ [\texttt{not}]\ \texttt{be}\ ([\texttt{in}]\ \texttt{the}\ \mathit{rset} \mid \texttt{empty})\\
\mathit{lset} &::= \texttt{every}\ (\mathit{entity} \mid \mathit{attribute})\ \{\mathit{attribute}\}\\
&\mid \texttt{the}\ \mathit{attribute}\ \{\mathit{attribute}\}\ \texttt{of the}\ \mathit{lset}\\
\mathit{rset} &::= \texttt{the}\ (\mathit{entity} \mid \mathit{attribute})\ \{\mathit{attribute}\}\\
&\mid \texttt{the}\ \mathit{attribute}\ \{\mathit{attribute}\}\ \texttt{of the}\ \mathit{rset}\\
\mathit{entity} &::= \mathit{noun},\ \mathit{attribute} ::= \mathit{adjective} \mid \mathit{noun} \mid \mathit{verb}
\end{aligned}$$

We allow possessive forms to be constructed both with the possessive apostrophe or preposition `of`. Likewise to structural boilerplates, the grammar describes the actual boilerplates as passed to the translation layer. For instance, the usage of the possessive apostrophe is irrelevant for the translation to Alloy and, although enforced by PROVA, is omitted from the grammar and erased prior to translation. For improved readability, we also allow an attribute to be followed by its target entity, but again these elements are erased before translation. Reference to a capability in a behavioral boilerplate should resort to the past participle, this being also replaced by the original verb before translation. Behavioral requirements can be translated to Alloy as follows:

$$\begin{aligned}
[\![l\ \texttt{shall be}\ r]\!] &\equiv \texttt{fact}\ \{\ \texttt{all t : Time},\ [\![l]\!]^{\epsilon} = [\![r]\!]\}\\
[\![l\ \texttt{shall be in}\ r]\!] &\equiv \texttt{fact}\ \{\ \texttt{all t : Time},\ [\![l]\!]^{\epsilon}\ \texttt{in}\ [\![r]\!]\}\\
[\![l\ \texttt{shall not be}\ r]\!] &\equiv \texttt{fact}\ \{\ \texttt{all t : Time},\ [\![l]\!]^{\epsilon}\ ! = [\![r]\!]\}\\
[\![l\ \texttt{shall not be in}\ r]\!] &\equiv \texttt{fact}\ \{\ \texttt{all t : Time},\ [\![l]\!]^{\epsilon}\ \texttt{not in}\ [\![r]\!]\}\\
[\![l\ \texttt{shall be empty}]\!] &\equiv \texttt{fact}\ \{\ \texttt{all t : Time},\ [\![l]\!]^{\texttt{no}}\}\\
[\![l\ \texttt{shall not be empty}]\!] &\equiv \texttt{fact}\ \{\ \texttt{all t : Time},\ [\![l]\!]^{\texttt{some}}\}\\
[\![\texttt{the}\ a_1\ \ldots\ a_l\ \texttt{of every}\ l]\!]^m &\equiv [\![\texttt{every}\ l\ a_1\ \ldots\ a_l]\!]^m\\
[\![\texttt{the}\ b_1\ \ldots\ b_n\ \texttt{of the}\ r]\!] &\equiv [\![\texttt{the}\ r\ b_1\ \ldots\ b_n]\!]\\
[\![\texttt{every}\ e\ a_1\ \ldots\ a_l]\!]^m &\equiv x : e\ ,\ y : x.[\![a_1]\!].\ldots.[\![a_{l-1}]\!] \mid m\ y.[\![a_l]\!]\\
[\![\texttt{every}\ a\ a_1\ \ldots\ a_l]\!]^m &\equiv y : \texttt{univ}.[\![a]\!].[\![a_1]\!].\ldots.[\![a_{l-1}]\!] \mid m\ y.[\![a_l]\!]\\
[\![\texttt{the}\ e\ b_1\ \ldots\ b_r]\!] &\equiv x.[\![b_1]\!].\ldots.[\![b_r]\!]\\
[\![\texttt{the}\ b\ b_1\ \ldots\ b_r]\!] &\equiv \texttt{univ}.[\![b]\!].[\![b_1]\!].\ldots.[\![b_r]\!]\\
[\![a]\!] &\equiv \begin{cases} a & \text{if } a \text{ immutable}\\ a.\texttt{t} & \text{otherwise}\end{cases}
\end{aligned}$$

Translation of possessives formed with `of` are reduced to the case of sequences of attributes separated by the possessive apostrophe. For the moment, we only allow the specification of invariant behavior. Thus, all generated facts begin with an universal quantification over `Time`. Mutable relation identifiers will then be projected over this quantified variable. Sequences of attributes are essentially translated using composition. Should the lhs sequence begin with an entity an universally quantified variable `x` is introduced in the context to be reused in the translation of the rhs if the same entity is again referred to (although our context free grammar does not mention it, a different entity cannot be mentioned in the rhs). An example of such behavioral requirement is the following on channels of a partition kernel:

$$\begin{array}{l}
[\![\texttt{The } \mathit{destination} \texttt{ of every } \mathit{Channel'}\texttt{s } \mathit{message}\texttt{s shall be} \\
\quad \texttt{the } \mathit{partition} \texttt{ of the } \mathit{Channel'}\texttt{s } \mathit{destination}]\!] \equiv \\
\texttt{all t : Time} \mid \texttt{all x : } \mathit{Channel}, \texttt{ y : x.}(\mathit{message}.\texttt{t}) \mid \\
\quad \texttt{y.}\mathit{destination} = \texttt{x.}\mathit{destination.partition}
\end{array}$$

If the lhs does not begin with an entity, then we just get the all range of the relation modeling the first attribute by precomposing it with the universal set $univ$. To translate requirements that test the cardinality of a set, the translation of the lhs receives an extra argument (in superscript) denoting the the multiplicity test that should be inserted. If the boilerplate is an inclusion (or equality) test, this parameter is set to nil. A possible example is

$$\begin{array}{l}
[\![\texttt{The } \mathit{channel} \texttt{ of every } \mathit{sent\ Message} \texttt{ shall be empty}]\!] \equiv \\
\texttt{all t : Time} \mid \texttt{all x : univ.}(\mathit{send}.\texttt{t}) \mid \texttt{no x.}(\mathit{channel}.\texttt{t})
\end{array}$$

Notice the usage of the target entity of the capability to enhance the sentence. As explained above, this is erased prior to the translation by PROVA.

4 Summary and Current Work

It is largely recognized by industry that the price of correcting an error grows exponentially in later life-cycle stages and that detecting and correcting errors at the requirements stage is of very high value. PROVA offers the translation from boilerplated requirements to Alloy. We claim that Alloy's *navigational style* offers a language with greater proximity to natural language than, e.g., first order logic. Ongoing work includes further development of boilerplates for dynamic behavior, namely through boilerplates that constrain valid traces in Alloy and correspond directly to LTL formulæ in Temporal Alloy.

References

1. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.
2. M. Elizabeth C. Hull, K. Jackson, and J. Dick. *Requirements engineering*. Springer, 2005.
3. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge Mass., 2006. ISBN 0-262-10114-9.
4. D.A. MacKenzie. *Mechanizing Proof: computing, risk, and trust.* MIT Press, 2001.
5. R. Ramos, E. Piveta, J. Castro, J. Araujo, A. Moreira, P. Guerreiro, M. Pimenta, and R. T. Price. Improving the quality of requirements with refactoring. In *VI Simposio Brasileiro de Qualidade de Software - SBQS2007, Porto de Galinhas*, 2007.
6. C. Videira and A. R. da Silva. Patterns and metamodel for a natural-language-based requirements specification language. In O. Belo, J. Eder, J. F. Cunha, and O. Pastor, editors, *CAiSE Short Paper Proceedings*, CEUR Workshop Proceedings. CEUR-WS.org, 2005.
7. K. S. Wasson. *Clear requirements: improving validity using cognitive linguistic elicitation and representation*. PhD thesis, Charlottesville, VA, USA, 2006. Adviser-Knight, J.
8. W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt. Automated quality analysis of natural language requirement specifications. In *Proceeding of the PNSQC Conference*, 1996.