The Magic of Specifications and Type Systems

Amin Bandali, Simon Hudon, Jonathan Ostroff
Software Engineering Lab, EECS

Introduction

- Architects draw detailed plans before a brick is laid or a nail is hammered. Programmers and software engineers don't. *Can this be why houses seldom collapse and programs often crash?*
- Blueprints help architects ensure that what they are planning to build will work. "Working" means more than not collapsing; it means serving the required purpose. Architects and their clients use blueprints to understand what they are going to build before they start building it.
- But few programmers write even a rough sketch of what their programs will do before they start coding.
- **Specifications**: To designers of complex systems, the need for formal specifications should be as obvious as the need for blueprints of a skyscraper. But few software developers write *specifications* because they have little time to learn how on the job, and they are unlikely to have learned in school. Some graduate schools teach courses on specification languages, but few teach how to use specification in practice. It's hard to draw blueprints for a skyscraper without ever having drawn one for a toolshed.

[*Leslie Lamport, Turing Award Winner, 2013*]

Specifications (and formal methods) used to be relegated to safety critical systems like nuclear power, avionics and medical devices. Increasingly, a variety of industrial strength formal methods (e.g. TLA$^+$ [4], Event-B [1], and many others) are now being used by Microsoft, Amazon, Facebook and Dropbox.

Significance & Contributions

Unit-B [3] is a new framework for specifying and modelling systems that must satisfy both safety and liveness properties.Compared to Event-B, Unit-B brings in record types and complete well-definedness. In comparison to TLA$^+$, Unit-B adds type checking, well-definedness checking and quantification over infinite sets.

Unit-B Web makes the Literate Unit-B prover available on the web. Unit-B Web leverages the automated predicate prover to two purposes:

- **Teaching**: can be used in classroom for demonstrations, or in evaluation in the form of online quizzes.
- **Online Proof Environment**, making specifications more accessible to casual users. It also serves as a "proof of concept" for a web IDE for the full modelling capabilities of Unit-B.

Unit-B Web's technology stack: • Syntax: based on LaTeX • Web: JavaScript, JSON, Yesod / Haskell • Prover: Haskell, Z3

Unit-B Web Snapshot

Below are two screenshots of the Unit-B Web tool, showcasing its type checking and well-definedness checking capabilities.

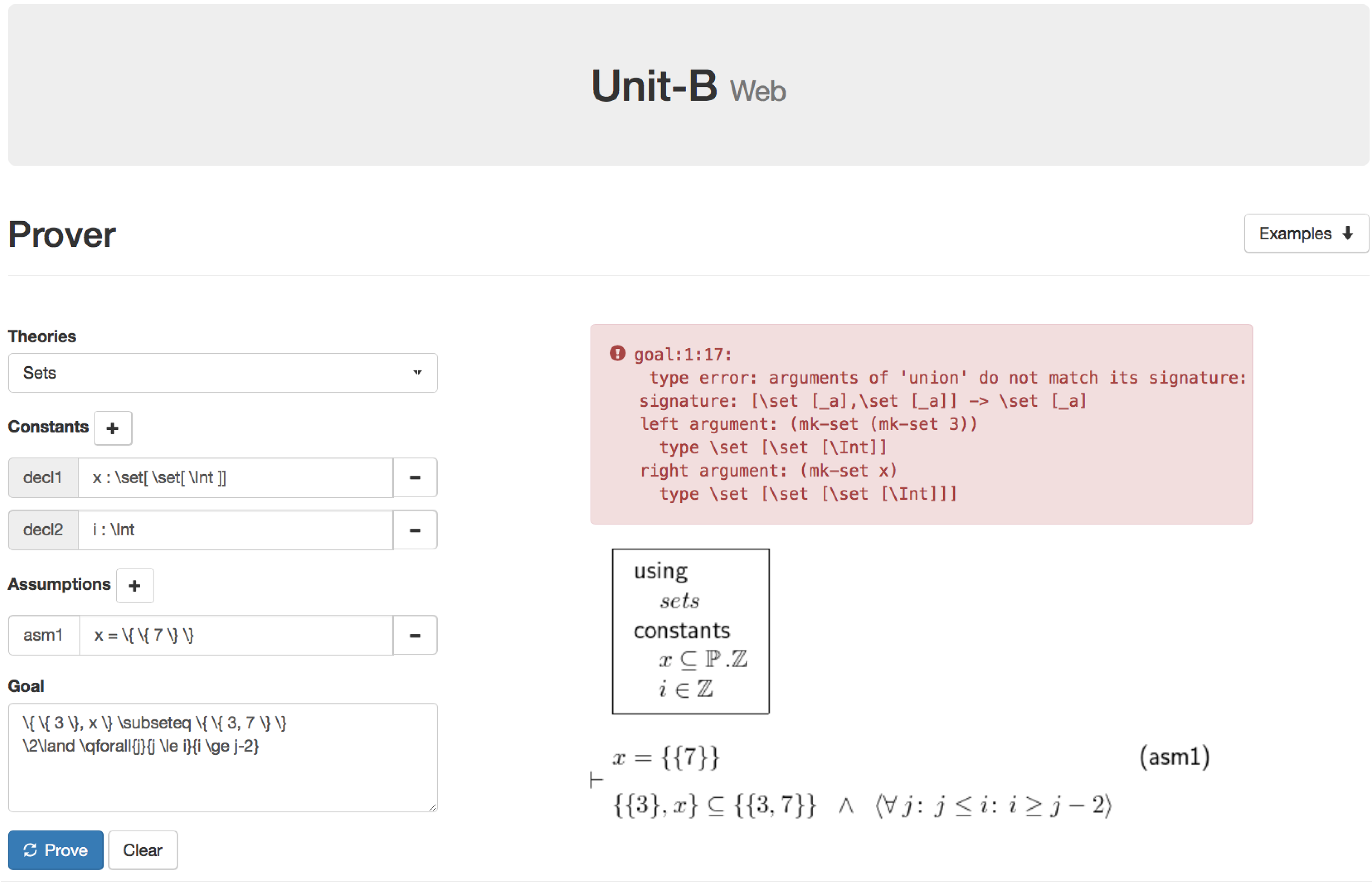

Figure 1: A type error — x is expected to be a set of numbers

Unit-B Web Snapshot

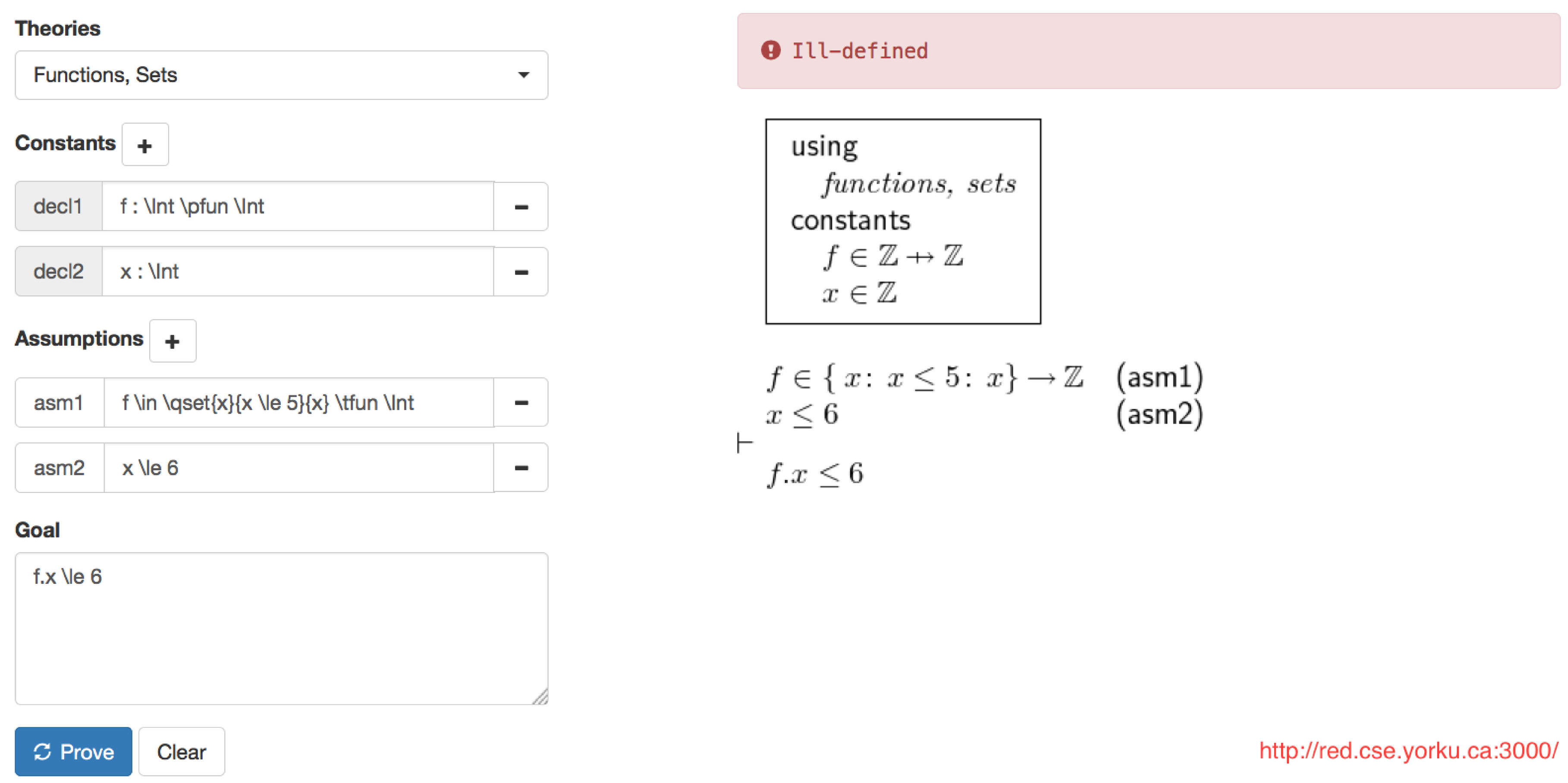

Figure 2: An ill-defined predicate — x is not in the domain of f

Type Checking

Some formulas, such as $\{x\} + 3 \le 7$, are not meaningful. Type checking helps us identify and fix them instead of laboring needlessly over the proof of meaningless formulas. TLA$^+$ does not recognize this as an error; Unit-B does (see Fig. 1).

TLA$^+$ is an untyped logic which allows expressive formulas such as $\{3, \{7\}\}$. In a simple type system such as that of Event-B, homogeneity rules out such a formula. The challenge in a typed system such as Unit-B is to allow such formulas. We do this using subtyping.

Further, type variables in Unit-B allow for polymorphic definitions, e.g. using the same functions on sets of numbers and sets of sets of numbers.

Well-definedness Checking

WD checking [2] catches meaningless formulas that the type checker cannot catch, such as division by zero or array out of bounds.

Unit-B's WD-calculus is complete; while Event-B's is incomplete. Let us consider the following example with four propositions A, B, C, and D (which we will specify shortly), where

$$A \Rightarrow WD(B)$$
$$B \Rightarrow WD(C)$$
$$B \Rightarrow WD(D)$$

The following calculation is not well-defined in Event-B (the red formula is rejected), but it is well-defined in Unit-B:

$$\begin{array}{ll} & A \wedge B \wedge (C \vee D) \\ = & \{associativity\} \\ & A \wedge (C \vee D) \wedge B \\ = & \{distributivity\} \\ & ((A \wedge C) \vee (A \wedge D)) \wedge B \end{array}$$

where

$$\begin{array}{l} A : x \in dom.f \\ B : f.x \in dom.g \\ C : g.(f.x) \le 3 \\ D : 7 \le g.(f.x) \end{array}$$

References

[1] Jean-Raymond Abrial.
Modeling in Event-B - System and Software Engineering.
Cambridge University Press, 2010.

[2] Ádám Darvas, Farhad Mehta, and Arsenii Rudich.
Efficient well-definedness checking.
In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pages 100–115, 2008.

[3] Simon Hudon, Thai Son Hoang, and Jonathan S. Ostroff.
The Unit-B method: refinement guided by progress concerns.
Software & Systems Modeling, pages 1–26, 2015.

[4] Leslie Lamport.
Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.
Addison-Wesley, 2002.