Formal Verification Toolkit for Requirements and Early Design Stages

Julia M. Badger[1] and Sheena Judson Miller[2]

[1] NASA Johnson Space Center, Houston, TX 77058, USA
[2] Barrios Technology, Houston, TX 77058, USA

Abstract. Efficient flight software development from natural language requirements needs an effective way to test designs earlier in the software design cycle. A method to automatically derive logical safety constraints and the design state space from natural language requirements is described. The constraints can then be checked using a logical consistency checker and also be used in a symbolic model checker to verify the early design of the system. This method was used to verify a hybrid control design for the suit ports on NASA Johnson Space Center's Space Exploration Vehicle against safety requirements.

1 Introduction

Requirements are a part of every project life cycle; everything going forward in a project depends on them. Good requirements are hard to write, there are few useful tools to test, verify, or check them, and it is difficult to properly marry them to the subsequent design. In fact, the inconsistencies and errors in the requirements contribute greatly to the cost of the testing and verification stage of the project. For example, over 75% of the errors in flight software are introduced in the requirements and design stage, but most are found in the testing, verification, and maintenances stages of the project, when it costs 10-100 times more to correct them [1].

This paper outlines a concept that formalizes the requirements and early design process by enabling verification during the requirements and early design stages of development and by linking the two stages together. Using natural language processing (NLP) tools and techniques combined with formal methods such as model checking to verify requirements and early design, it is possible to reduce the number of errors introduced during these stages, thus reducing the overall software and project cost.

Requirements engineering is a difficult field because of the relative lack of constraints on the requirement generator, which makes errors and inconsistencies likely. Several groups have tried to use formal methods to verify sets of requirements. Current methods require NL requirements to be manually translated into a model or formal specification language for model checking. A specification notation called Software Cost Reduction (SCR) that is loosely based on event-driven logic can be used to find invariants in a model of the system described by the requirements (or mode machine) [2]. SCR has been translated to a format

that works with Symbolic Model Verifier (SMV) and used with several projects [3, 4]. A similar project that converted requirements written in Formal Tropos into Promela and verified them using SPIN was described in [5].

In this method, formal specifications in the form of logical statements are automatically created from sets of natural language (NL) requirements. The products that result from NLP of requirements are used in three ways. First, the specifications are checked for consistency by a novel set of tools that can convert and formally verify logical formulas. Second, the state space automatically derived from the requirements can be used by system designers during the early design stage to create a linear hybrid system model for the system [6]. Third, this model is verified against the formal specifications that were created from the requirements using a symbolic model checker, such as InVeriant [7].

A more complete description of the components of the tool chain can be found in Section 2. A small example of this design and verification method on the suit port design for the Space Exploration Rover (SEV) is described in Section 3 and Section 4 concludes the paper.

2 Tool Chain Description

The tool chain for this concept was developed from extensions to several existing tools, such as the STAT (Semantic Text Analysis Tool) natural language parser, the SBTChecker (State Based Transitions Checker), a hybrid system design tool, and InVeriant, a symbolic model checker. This tool chain also incorporates a novel tool that could be derived from several existing capabilities. A figure describing the tool chain is shown in Fig. 1.

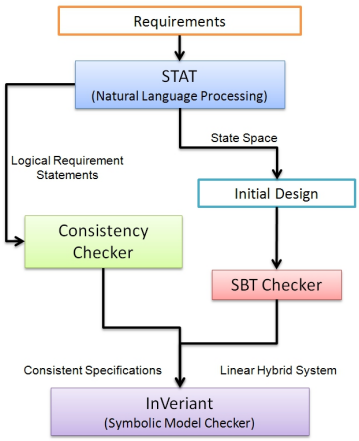

Fig. 1: Tool chain description.

2.1 STAT

STAT was developed for analyzing problem reports, in which critical information is contained within data-fields of English-language sentences. STAT consists of a statistical parser, the Stanford Parser [8], which annotates a sentence; an algorithm, the minimal clausal reconstruction algorithm, that recognizes and fills empty categories in the parser's output [9]; and a semantic interpreter that uses a lexical Aerospace Ontology [10].

This work uses STAT's capability to parse and scope English-language sentences from a set of requirements into formal logic. The STAT software structures and annotates the text, and then produces the formal logic statements using a logic conversion module. STAT also has the capability to find state variables from the subjects of the sentences and a partial state space from the predicates.

2.2 Logical Consistency Checker

The logical consistency checker will be used to collect logical requirements and verify that they are all consistent given the appropriate state space rules. There are several existing tools that may be leveraged. SALT [11], for example, is a tool that creates logical statements from a highly specialized language that could be leveraged as an intermediate step between the STAT output and a formal LTL statement. Also, Spin [12] is capable of converting LTL formula to ω-automata; it could convert the logical statements into a verifiable form and check the composed automata for consistency and logical errors. Novel work is concentrating on dealing with the inevitable state space explosion.

2.3 SBTChecker

The SBTChecker is a design tool that helps one create hybrid systems that have state-based transitions over a given passive (or uncontrolled) state space. Having state-based transitions is a property that significantly improves the ability to verify a system by reducing the effect of state space explosion [7].

Definition 1. *Let $\mathcal{D} = \{d_1, d_2, ..., d_n\}$ be the set of passive state variables. Then, let Γ be the passive state space, $\Gamma = \Lambda_1 \times \Lambda_2 \times ... \times \Lambda_n$. If for each state $\gamma_i \in \Gamma$, there exists some location, $v_j \in V$, in the set of locations such that the passive state satisfies the invariant, $\gamma_i \models inv(v_j)$, and each transition condition, $\sigma_k \in \Sigma$, is based only on the passive states of the system, then the hybrid automaton has* state-based transitions*.*

A hybrid system is a set of concurrently executed hybrid automata. The SBTChecker can then be used to design individual automata that will be combined into a concurrent hybrid system.

Theorem 1. *If all controlled constraints on each state variable $x_i \in X$ in each location $v_j \in V_n$ are consistent, that is, able to be executed concurrently or merged, and if each automaton in the set of hybrid automata, $H_n \in \mathcal{H}, n = 1, ..., N$, has state-based transitions over $\mathcal{D}_n \subseteq \mathcal{D}$, then the composition, $H' = H_1 \circ ... \circ H_N$, has state-based transitions over $\mathcal{D}$.*

2.4 InVeriant

The InVeriant software creates the locations and invariants from the set of concurrently-executed hybrid automata, checks consistency, and composes it with the unsafe set constraints to find unsafe locations. Then, based on the following theorem, the path to these unsafe locations depends only on the paths to the corresponding passive state values.

Theorem 2. *Given a hybrid system with a set of locations* $V_k = \{v_1, ..., v_n\}$ *whose transitions are based on the discrete states of a set of passively-constrained state variables* $\mathcal{D}_k = \{d_1, ..., d_m\}$*, if all the discrete states associated with each passive state variable are reachable from each other, then all locations* $v_l \in V_k$ *are reachable from any other location,* $v_j \in V_k$*.*

The unsafe set is a collection of disjoint sets of constraints, $Z = \{\zeta_1, ..., \zeta_n\}$, where each disjoint set of constraints, $\zeta_i = \{z_1^i, ..., z_{n_i}^i\}$, has separate constraints on individual state variables, and each separate constraint $z \in (X^d \cup \dot{X}^c \cup \mathcal{D}) \times Q \times (\mathbb{R} \cup \Lambda)$ constrains a discrete controllable state variable (X^d), the rate of a continuous controllable state variable ($\dot{X}^c$), or a passive state variable. The verification algorithm simply composes the hybrid system's locations with the unsafe set descriptions and checks the composition for consistency. If the composition is valid, InVeriant takes steps to determine the reachability of the location, which is simplified by the state-based transitions property and Theorem 2.

3 Preliminary Results

The concept verification procedure was implemented on the suit port state machine for the SEV. The SEV is a next-generation modular concept vehicle intended to be used in a variety of target environments including the moon, Mars, or even an asteroid. The current generation consists of an enclosed cabin mounted on a wheeled chassis. The cabin is designed to allow two crew members to live and work in shirt sleeves for up to two weeks; two spacesuit ports enable the crew to easily exit the cabin for extra-vehicular activity (EVA).

The suit port consists of an inner and outer hatch (the inner one is actuated, the outer one is connected to the suit), a seal around the suit (PLSS) and an actuated Marmon to secure the suit in place. The vestibule is the area between the inner and outer hatches. The passive state variables and their corresponding state values are listed in Table 1.

The transitions between the states in the passive state space are all considered to be stochastic with trivial models (valid transitions occur between all states). There are two exceptions; the first is the pressure status state variables, which have linear discrete models that are based on real continuous values. The second exception is the Command state variable; the model for this is shown in Fig. 2. This model is generated from the requirement that it must be possible to safely initiate ingress or egress from any point in the procedure.

The first design of the hybrid control state machine for the suit ports was designed using the SBTChecker, and therefore has state-based transitions. The

Table 1: Passive State Variables

State Variable	Abbreviation	States
Suit/Vestibule Pressure Status	SP/VP	EVACUATED, LOW, EQUALCABIN, HIGH
Suit Status	SS	ATTACHED, DETACHED
Hatch/Marmon Health	HH/MH	NOMINAL, JAMCLOSED, JAMOPEN
Hatch/PLSS Seal/Vestibule Leak	HL/PL/VL	VERIFIED, FAILED, UNKNOWN
Command	CM	HOPEN, HCLOSE, MOPEN, MCLOSE

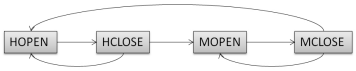

Fig. 2: Model for the Command state variable for the suit port verification example

flow of a nominal egress would follow the path shown in Fig. 3 a; the flow of a nominal ingress would follow the path shown in Fig. 3 b.

Four natural language safety requirements were run through STAT. These requirements were the following:

1. The hatch shall remain closed unless the Marmon is closed, the PLSS seal is verified, and the vestibule pressure is equalized with cabin.
2. The Marmon shall remain closed unless the hatch is closed, the hatch seal is verified, and the vestibule is evacuated.
3. The hatch shall be closed whenever the suit is detached.
4. The hatch must be closed whenever the suit pressure is not equalized with the cabin.

The STAT tool output the requirements in xml form, which were used directly in the input file for the InVeriant model checker. The state machine and the safety requirements were run through InVeriant, which found several places where the unsafe set as specified by the safety requirements was reachable. Both the requirements and the hybrid control system were redesigned and the new design and requirements were verified against one another to ensure quality and correctness.

4 Conclusions and Future Work

The formal early design and verification concept presented here has many benefits. Automatically converting natural language requirements into a formal spec-

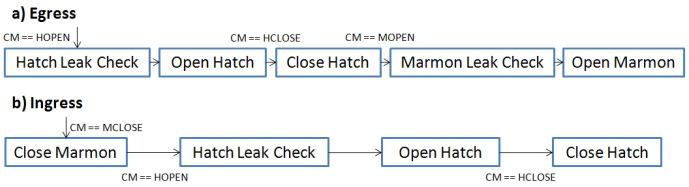

Fig. 3: Nominal execution flow for a) egress and b) ingress

ification that can be checked for consistency and subsequently used in a verification algorithm will reduce the number of errors introduced in the requirements and early design stages. Likewise, formally tying the requirements and early design together will ensure a better design. Currently, work is underway on adapting STAT to pull state space information from natural language requirements, on leveraging tools to design a logical consistency checker, and on understanding the classes of requirements and systems that can use this method. This concept will be tested on sets of requirements for other flight products at NASA-JSC, including the SAFER project.

5 Acknowledgements

The authors would like to acknowledge David Throop for his work on the STAT extensions for the suit port example, Jane Malin for invaluable discussions and mentoring, and Aaron Hulse for sharing his suit port state machine for the verification example.

References

1. D. Peercy, *Software Quality Engineering Course Guide*. SEMATECH, 1995.
2. R. Jeffords and C. Heitmeyer, "Automatic generation of state invariants from requirements specifications," in *Proc. 6th Int'l Symp. on Foundations of Software Engineering*, Nov. 1998.
3. T. Sreemani and J. M. Atlee, "Feasibility of model checking software requirements: A case study," in *Proc. 11th Conf. on Computer Assurance*, pp. 77–88, IEEE, 1996.
4. W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese, "Model checking large software specifications," *IEEE Transactions on Software Engineering*, vol. 24, pp. 498–520, July 1998.
5. R. Kazhamiakin, M. Pistore, and M. Roveri, "Formal verification of requirements using SPIN: A case study on web services," in *Proc. of 2nd Int'l Conf. on Software Engineering and Formal Methods*, 2004.
6. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HyTech: A model checker for hybrid systems," *International Journal on Software Tools for Technology Transfer*, 1997.
7. J. M. B. Braman, *Safety Verification and Failure Analysis of Goal-Based Hybrid Control Systems*. PhD thesis, California Institute of Technology, 2009.
8. D. Klein and C. D. Manning, "Fast exact inference with a factored model for natural language parsing," in *In Proc. of Advances in Neural Information Processing Systems 15 (NIPS*, pp. 3–10, MIT Press, 2003.
9. J. T. Malin, C. Millward, F. Gomez, and D. R. Throop, "Semantic annotation of aerospace problem reports to support text mining," *IEEE Intelligent Systems*, vol. 25, pp. 20–26, 2010.
10. J. T. Malin and D. R. Throop, "Basic concepts and distinctions for an aerospace ontology of functions, entities and problems," in *Proc. of IEEE Aerospace Conference*, March 2007.
11. A. Bauer and M. Leucker, "The theory and practice of SALT," in *Proc. of 3rd NASA Formal Methods Symposium*, April 2011.
12. G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.