Challenges in verification and validation of autonomous systems for space exploration

Guillaume Brat and Ari Jonsson
USRA/RIACS

Abstract—Space exploration applications offer a unique opportunity for the development and deployment of autonomous systems, due to limited communications, large distances, and great expense of direct operation. At the same time, the risk and cost of space missions leads to reluctance to taking on new, complex and difficult-to-understand technology. A key issue in addressing these concerns is the validation of autonomous systems. In recent years, higher-level autonomous systems have been applied in space applications. In this presentation, we will highlight those autonomous systems, and discuss issues in validating these systems. We will then look to future demands on validating autonomous systems for space, identify promising technologies and open issues.

1. INTRODUCTION

Space exploration is a powerful driver for technological advances. To find answers to questions unanswered by past missions, space missions continually become more complex, go further and longer, and provide more involved scientific observations. One aspect of the technological improvements is the role of human control in operating spacecraft. Earth-observing satellites and lunar excursions are easily controlled from Earth, albeit at the expense of maintaining large operations staff. More distant missions, such as those to Mars and the outer planets, make direct human operations more difficult and expensive, due to communications time lag and the consequent need for the spacecraft to operate for long times without supervision. Some mission concepts are impossible with complete reliance on human operations; e.g., extended exploration of moons orbiting outer planets.

The use of autonomy in space missions will greatly benefit space exploration. First, the operations cost can be reduced significantly as fewer human operators are required; an essential aspect of making future exploration missions cost effective and sustainable. Secondly, the use of autonomy will permit spacecraft to achieve more in less time, and to accomplish what is otherwise not possible. For example, autonomous Mars rovers could respond to interesting events like dust storms, while autonomous lunar construction rovers could operate continuously. However, autonomous systems are still rarely used in space exploration; this is in part due to reluctance to change and in part due to a technology gap in certifying the behavior of such systems.

The cost of space missions is extremely high. Cheap Mars missions cost hundreds of millions of dollars. For example, the Mars Surveyor missions in 1998 (which consisted of an orbiter, Mars Climate Orbiter, and a lander, Mars Polar Lander) cost over $300 millions. Expensive missions, such as Cassini, can cost billions of dollars. Moreover, the risk of failure for these missions is still very high. For example, even though the systems used in the Mars Surveyor missions were fairly simple, both missions were lost. It is therefore not surprising that mission managers take a conservative approach when designing their missions and systems. The first success criterion is to reach the target (be it Mars or Titan); then come scientific goals. In this context, it is not surprising that the credo of mission managers is to *fly what has been successfully flown before*. This leaves little opportunities for incorporating new technologies, unless they come with strong guarantees about safety and reliability. In the case of autonomous systems, this gives rise to a very challenging verification and validation problem.

By nature, autonomous systems are required to be more flexible than traditional systems. Their strength is in adapting to a large set of circumstances, some of which may be unknown at design time. This strength becomes their main weakness when it comes to validation. Our current validation techniques struggle with existing mission systems and now we are faced with validating autonomous systems that can exhibit a much larger set of behaviors. In fact, even the implementation of these systems presents a new challenge. In the past, the implementation of embedded systems has been quite conservative. For example, dynamic memory allocation is allowed only at starting time as a pool of available memory, and never at run time. This guarantees that the memory consumption stays bounded. This has strong implications in terms of the data structures used in these systems; e.g., arrays being preferred to dynamic lists. Such restrictions make systems somewhat amenable to advanced automated verification techniques. Hardly any such "useful" restrictions hold for autonomous systems.

So, should we just give up on validating autonomous systems? We certainly should not. The need for autonomy in future missions is clear. In addition, inroads have recently been made in limited use of autonomy-based software for missions. For example, the MAPGEN tool is being used in the context of the Mars Exploration Rover (MER) mission. MAPGEN is a planning and scheduling system which is used to prepare the plans that are up loaded

to the rovers at the beginning of each Martian day. Before MAPGEN, this task was essentially done by hand, severely limiting the trade-off analysis that could be done. The use of MAPGEN has allowed the MER ground control team to reduce the human involvement, speed up the construction of these sequences, and increase science return. Despite its obvious advantages, MAPGEN was not easily accepted by mission managers. One of the critical obstacles was the validation of the system, and while a careful testing process satisfied the MER mission's specific requirements, it was clear that a better validation process was needed. This observation is valid for any autonomous system that sees itself as a candidate for being on-board a spacecraft or a rover. The requirements and operational limits placed on autonomous spacecraft software, such as the Remote Agent Experiment and the EO-1 Flight Experiment, demonstrate this [15].

Fortunately, researchers from the Artificial Intelligence and Software Engineering communities are recognizing the need for good validating techniques. Recently, the research community has focused on exploring the commonalities between automated validation and verification (V&V) and model-based computing. So far, this activity has yielded little practical results, but new efforts are underway. In this paper, we discuss our approach to tackling the validation and verification of autonomous decision-making control systems, and highlight some of the techniques being applied.

The goal of our work is to go beyond exploration and trials, and move on to develop practical techniques. Our approach is as follows. First, since models are at the core of autonomous systems, we need to develop techniques to verify models, constraints and their manipulations in reasoning engines. For that, we plan on relying on model checking techniques, which have matured considerably the past few years. This allows us to verify safety properties on autonomous systems. In addition, our experience with previous NASA software shows that run-time errors (e.g., out of bound array accessing, or arithmetic overflow and underflow) are the most common programming errors. Therefore, we propose to complement model checking with the use of static analysis. Recent work has shown that modern static analyzers are not plagued with problems of scale and the generation of too many false alarms. Yet, we believe that scalability will remain an issue since autonomous systems yield such large sets of behaviors. We tackle this problem from two angles. First, we will rely on the assume-guarantee framework to implement a form of compositional verification. This allows us to divide the verification process into two phases. In the first phase, modules are verified separately. In the second phase, verification results for the modules are put together to ensure the verification of the whole system. Recent work has shown the feasibility of the approach to verify the executive a planning and scheduling system. Second, we will investigate the automatic synthesis of code for autonomous systems. Code synthesis is an interesting and useful approach, as it reduces the role played by human in code development, while also providing an opportunity for controlling the "shape" of the code for autonomous systems. This capability is extremely interesting in terms of verification because it allows us to bias our verification algorithms towards specific coding constructs. This typically leads to more precise and scalable verification.

2. AUTONOMOUS SYSTEMS

Most autonomous systems use a model-based approach. In a model-based system, application-specific information, like the capabilities and limitations of a rover, are described in a model of the application, not in implemented code. A general-purpose reasoning system uses the model to operate the entity in question. To ground this, consider a simple Martian rover. A model of this rover would describe its operational capabilities and limitations at a suitable level of abstraction; for example, it would specify that the rover can take a picture in a given range of directions, that taking such a picture requires the camera and CPU to be powered on, and that the taking of a picture will consume given quantities of energy and data storage resources. A general-purpose model-based autonomous system would then use such a model to operate the rover. It would make sure that planned operations stay within energy and data storage capacity limit and that the execution of planned operations yields the expected results. Systems operating rovers, spacecraft, and other systems in such a plan-execute cycle are very common.

Model-based autonomus planning and execution systems are the primary candidates for future roles in spacecraft operations and space exploration. Consequently, we focus our V&V efforts on these systems. To ground our examination, we will look at an autonomy architecture currently under development in a collaborative effort between researchers at NASA Ames Research Center and NASA's Jet Propulsion Laboratory. The architecture is based on a robotic control framework called CLARAty (Coupled Layer Architecture for Robotic Autonomy). CLARAty [16] is a modular architecture, designed specifically for space-based robotic control applications. It provides a wide-range of robotic control functionality and is designed to ease the integration of new technologies on robotic platforms. CLARAty features a *Functional Layer* of robotic primitives, coupled with a *Decision Layer* of planning and execution functionality; each of these layers contains a hierarchy of components ranging from the most elementary to the most "intelligent".

The *Functional Layer* (FL) provides a set of standard, generic robot capabilities that interface to system hardware [16]. These capabilities are organized as a software class hierarchy of robotic components; for example, wheeled-mobility is a subclass of mobility, and individual rover wheel assemblies are child classes. As is natural in object

oriented systems, the interface is separated from implementation. Physical limitations of devices are distinguished from algorithmic limitations. Finally, runtime models of devices are incorporated in the Functional Layer.

The *Decision Layer* (DL) [6] provides capabilities for autonomously creating and executing sequences of rover actions that achieve specified operations tasks. This involves planning, scheduling, monitored execution, and a number of other capabilities. In the past, the decision layer capability has been provided by the CASPER planning system [4] and the TDL executive [18], which were developed at JPL and CMU. TDL is a C++ extension that provides syntactic support for task decomposition, synchronization, execution monitoring and exception handling. CASPER is a continuous planning system that can continually update the current plan based on changing state, environment and mission goal information.

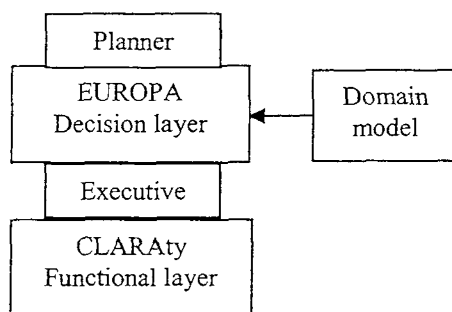

Figure 1. The autonomous system architecture.

In the autonomy architecture used in our work, the decision layer capabilities are provided by the Extendable Uniform Remote Operations Planning Architecture (EUROPA). The architecture of the integrated system is shown in Figure 1. EUROPA is a model-based planning and scheduling architecture descended from the Remote Agent Planner; previous versions of the technology are described in [9, 14]. Users of EUROPA can specify the rules of planning domains using a rich domain description language that supports time, resources, disjunctive preconditions and conditional effects. EUROPA provides support for "foreign function" calls implementing arbitrary constraints. EUROPA consists of a hierarchy of highly configurable components, supporting the development of many types of planners and plan representations. EUROPA's plan databases employ a generic constraint reasoning capability that can support both long-range deliberative planners as well as short-horizon continuous planners. Finally, EUROPA allows multiple planners to modify the same plan.

3. V&V CHALLENGES

The elements of the autonomous system we are considering offer different challenges in terms of verification. For example, the V&V issues with a planner are quite different from the V&V issues of a functional component controlling a physical device (such as the wheel of a rover). In the first case, V&V is dealing with issues such as correct manipulation of plans, domain model consistency checking, or correct resource utilization reasoning. In the second case, we are more concerned with traditional issues such as runtime errors, concurrency and timing problems. The V&V tools that can address these problems are quite different. Moreover, they are at different maturity levels. In any case here is a list of the challenges we are facing.

Functional layer

The functional layer consists of C++ device controllers organized in a hierarchy of controllers. Some are simple controllers; others are more complex and consist of several components cooperating to implement complex controllers (e.g., controllers for wheels include controllers for adjusting speed as well as for turning wheels). In general, they do not present a new challenge for V&V in the sense that they are similar to controllers in embedded systems. Moreover, since most of the "intelligence" in autonomous systems is placed in the decision layer, they are fairly simple C++ programs. Therefore, we do not anticipate any problem applying traditional V&V tools such as static analyzers.

The static analysis of functional components consists of verifying their implementations with respect to the formal semantics of C++. The main task is to certify that a component is free of errors such as array-out-bound accesses, null pointer de-referencing, and the use of un-initialized variables. These errors are violations of the C++ semantics, but are not yet checked by compilers. Fortunately, they can be checked at compile time using sophisticated static analyzers. These analyzers come in two categories. The first one consists of fast and highly scalable analyzers, which unfortunately do not guarantee full coverage (Coverity, Clockwork). These tools are typically useful in a development environment, but not so much in a certification context. The second category consists of analyzers guaranteeing full code coverage (PolySpace, CGS, Astrée) [1, 20]. We will investigate the synergy between these tools and the certification done in synthesis.

Furthermore, controllers are fairly small programs, which will allow us to check complex properties (such as response properties of the style "upon this event occurring, this action should be taken") using model checkers. Model checking is the task of verifying that a property (a functional correctness criterion) is a model of a system (the code implementing the functional component). Functional correctness properties are defined using logic or automata. In recent years, there has been quite a bit of interest in applying model checking to software verification [2, 5, 11, 12]. These experiments have demonstrated two important results. First, model checking is applicable directly to code rather than having to translate code to a model using some modeling language (which always introduces uncertainty with respect to the

correctness of the translation). For example, the JPF model checker works directly on Java byte code. There is on-going work to adapt this strategy to C++. Second, scalability remains a critical issue. However, this technology can scale to a few thousand lines of code, which is sufficient for our purpose.

Physical controllers present another major validation challenge that is not always addressed by the V&V community. Indeed, controllers generally assume specific behaviors for the physical systems they control. These assumptions may not correspond to the physical reality. This may be addressed by using high-fidelity testbeds, though it only postpones the problem to the validation of the testbed. Ultimately, there is no escaping field testing to verify that the behavioral assumptions fit reality. Even there, we are only able to approximate the conditions under which these applications will be deployed (e.g., Mars).

Decision layer

Verifying a planner is an enormous challenge considering that planners are meant to find intricate solutions in very large state spaces. Therefore, pretending to be able to verify the same state space with V&V tools seems somewhat futile. However, search techniques in V&V tools have one advantage, in this respect, over planning search techniques; they take advantage of abstraction. Typically, V&V tools focus on one property at a time, which allows them to focus their search on specific aspects of the state space while blurring the rest in a conservative (in the sense that they generalize behaviors into classes of behaviors rather than eliminating some behaviors) abstraction. In principle, search techniques in planning could achieve the same by using appropriate heuristics (i.e., the heuristic computes the abstraction dynamically). Incidentally, abstraction is also used to verify functional components. In any case, since the problem of verifying a planner is so complicated one might wonder what can be done to give us some assurance that the planner does its job.

First, we can look at the domain model used by the planner. Model checking techniques can be used rather efficiently to prove interesting properties on models. For example, model checkers can be used to check for inconsistencies, ambiguities, and, in some cases, incompleteness. Pecheur and Cimatti showed the feasibility of this approach by applying it to diagnosis systems [17]. There seem to be two complementary approaches. The first approach consists of translating the constraints in the domain model into SMV properties and then using the SMV symbolic model checker to verify the properties [3]. In general, symbolic model checking scales very well. This is especially true for the new generation of checkers based on SAT resolution. The second approach relies on explicit state model checkers (in which the state space is explored using DFS or BFS search algorithms) such as SPIN [13] or JPF. Pecheur showed that they can be used to diagnose ambiguities in models.

Second, we can actually verify that some basic planner capabilities are correctly implemented. Planners manipulate plans, which are nothing but list or tree data structures. In general, lists constitute a hard problem for static analyzers and model checkers. Part of the problem resides in the fact that they can grow unbounded. However, the fundamental problem is that, so far, they have not laid themselves well to abstractions. Therefore, advanced V&V tools are mostly used as advanced testing tools. For example, an explicit-state model checker is used to explore all concurrent aspects for a given test input. To realize that, it also has to closely approximate the system behaviors; this results in a loss of scalability and generality. Furthermore, these tools do not provide much benefit over the traditional testing paradigm since they still rely on generating appropriate (i.e., meaningful with respect to the application conditions and extensive in terms of behavioral or code coverage) test sets.

Fortunately, recent research has shown that it is possible to design efficient abstraction for some specific types of lists and trees. For example, Venet has designed a numerical abstraction for well-behaved (which we will left unspecified at this time) lists [19]. He uses this abstraction to design a static analysis that is precise, and scalable, for these lists. For example, the analysis can track order (e.g., causal) relationships amongst list elements. In another example, Visser and Pasareanu use a list abstraction that can be used to automatically generate test cases for programs manipulating complex data structures. Their technique propagates throughout the program a symbolic list representation that is partially concretized when individual fields (of the data structure) are accessed by the currently considered instruction. This "lazy" concretization strategy can be used to obtain test inputs that provide a specific coverage of the program. We believe that such techniques are getting mature enough that we can use them to provide meaningful V&V results on the planner.

Interface and System Composition

The problem of verifying an executive layer is not fundamentally different from verifying the decision layer. Again, we are facing the problem of verifying a program that manipulates lists or trees (i.e., plans). While a planner puts a list together according to activities, an executive checks timing issues and then executes the plans by calling the right functional elements at the right time. Again, we can use the same techniques (static analysis with numerical abstraction and symbolic execution with list abstraction) to evaluate the correctness of the executive. Yet, we are still left with the problem of verifying that the executive interfaces correctly with the planner and the functional elements.

Actually, the problem of composition is not restricted to the interface layer. Because of their internal complexity other elements face the same issue. For example, a functional element might consist of a composition of simpler functional elements. The planner itself might rely on

different modules to deal with different issues such as resource checking [8] or reasoning under uncertainty. Therefore, it is important to devise a V&V strategy that can deal with system composition. It promotes re-usability of components (and V&V artifacts) as well as V&V scalability. Fortunately, compositional verification has been the object of intense research these past few years. We intent to build on it and, for example, expand the work of Giannakopoulou and Pasareanu on assume-guarantee reasoning.

Compositional verification is a divide-and-conquer technique that aims at taking advantage of the modular architecture of a system in order to decompose the hard (expensive) problem of system verification into manageable verification of its components. The idea is to check separately each component against a local property, in order to avoid the state-space explosion problem associated with checking a global (system-level) property on the entire system; this problem occurs as a result of storing the entire system state in memory. Components often satisfy properties only in specific contexts (or environments). Assume-guarantee style compositional reasoning imposes that each component makes assumptions about its environment (i.e. the rest of the system). Of course, these assumptions need to be discharged by verifying that the rest of the system satisfies them. To automate assume-guarantee reasoning, appropriate assumptions must be generated, which can be done automatically from legacy code as described in [10].

5. THE ROLE OF SYNTHESIS

As we discussed in the previous section, system composition is a central issue in our work. In fact, it is the most important aspect of our work since our goal is to provide a framework allowing a fast, and trusted, reconfiguration of autonomous systems. For example, we look at scenarios where some physical elements (such as cameras on rovers) can be plugged in and out of the system. Another more extreme scenario consists of changing the planner or, at least, modifying the planning strategy. Our target users are users, and not designers, of this technology. Therefore, we want to automate the reconfiguration of the system. To accomplish this goal, we will heavily rely on software synthesis.

Software synthesis is the process of automatically generating programs from high-level specifications by successive steps. Software synthesis can be used to automatically generate programs to perform a desired function from a given software component library. The goal of certifiable synthesis is to separate the code generation from the notion of correctness of the code while still providing support for verification in the form of automatically generated annotations [7]. For example, Fischer *et al.* use techniques based on schema rewriting rules to generate both code and annotations, and, automated theorem proving and proof checkers to certify the generated code. Existing techniques use automated theorem proving (i.e. no user intervention, and therefore, no expertise required). Past experience shows that synthesis can be adapted to practical applications, even those involving complex mathematical specifications such as differential equations. Existing work has demonstrated realistic program synthesis in the domains of state estimation [21] and data analysis software. Other tools have been used to generate code functionally equivalent to the attitude control system of the Deep Space 1 spacecraft, as well as data analysis code for Hubble planetary data.

Moreover, the generation of annotations offers opportunities of synergy between synthesis and software verification techniques, such as

(1) pre-verification of the schemas – providing some support for modularization of the verification process,

(2) generation of certification conditions – allowing an earlier use of verification,

(3) availability of high-level contextual information – providing us with clues about the user intent,

(4) control over the coding constructs, – allowing us to pick the construct most amenable to verification, and,

(5) control over the structure of the generate code, – allowing us to reduce the state space explored by the verification engines.

Our strategy consists of placing synthesis at the center of the reconfiguration process. By controlling the structure of the code we generate we can limit the risks associated with using advanced V&V techniques (see previous section). Moreover, synthesis comes with a certification strategy that we can use to provide trusted systems.

6. CONCLUSION

In this paper, we presented a strategy to verify autonomous systems (in our case, planning and scheduling systems). We are relying on a combination of advanced verification techniques (static analysis, model checking, compositional verification, and automated test generation) and code synthesis (with certification) to provide trust in these types of systems. Our approach goes together with a design strategy in which we configure/design autonomous systems based on known building blocks (the CLARAty functional components and the EUROPA framework). Therefore, the notion of composition (in terms of both verification and system design) is critical to the success of our endeavor.

We believe that our approach is a good starting point to designing practical V&V techniques for autonomous

systems. However, we are very interested in the opinion of the research community. Therefore, we hope that this talk will be an opportunity to spark interest in the validation of autonomous systems. We are looking forward to the discussions that our approach is bound to create.

We would like to acknowledge the contributions of the other members of our project to this discussion. Jeremy Frank, Tara Estlin, and Mark Boddy provided invaluable information about P&S technology (in particular with regard to CLARAty and EUROPA). Dimitra Giannakopoulou and Ewen Denney were also precious in shaping our vision about verification and synthesis.

7. REFERENCES

[1] B. Blanchet *et al.* "Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software." LNCS 2566, pp. 85-108, 2003.

[2] G. Brat, S. Park, K. Havelund, and W. Visser, "Java PathFinder – Second Generation of a Java Model Checker," In POST-CAV Workshop on Advances in Verification, 2000.

[3] J. R. Buch, E.M. Clarke, K.L.McMillan, D.L. Dill, and J. Hwang, "Symbolic Model Checking: 10E20 states and beyond," In LICS, 1990.

[4] S. Chien, R. Knight, S. Stechert, R. Sherwood, and G. Rabideau, "Using Iterative Repair to Improve the Responsiveness of Planning and Scheduling," Proc of the Fifth Int'l Conf. on Artificial Intelligence Planning and Scheduling, April 2000.

[5] W. Deng, M. Dwyer, J. Hatcliff, G. Jung, Robby, and G. Singh, "Model-Checking Middleware-based Event-driven Real-time Embedded Software," Proceedings of 1st Intl. Symposium on Formal Methods for Components and Objects, 2002.

[6] T. Estlin, F. Fisher, D. Gaines, C. Chouinard, S. Schaffer, and I. Nesnas, "Continuous Planning and Execution for an Autonomous Rover," 3rd Int. NASA Workshop on Planning and Scheduling for Space, Oct 2002.

[7] B. Fischer and J. Schumann, "AutoBayes: A System for Generating Data Analysis Programs from Statistical Models. Journal of Functional Programming, Vol. 13, No. 3, May 2003, pp. 483-508.

[8] J. Frank, "Bounding the Resource Availability of Partially Ordered Events with Constant Resource Impact", In Proceedings of the 10th International Conference on Principles and Practices of Constraint Programming, 2004.

[9] J. Frank and A. Jonsson, "Constraint-Based Attribute and Interval Planning." In the Journal of Constraints, vol. 8, no. 4, 2003.

[10] D. Giannakopoulou, C. Paraseanu, and H. Barringer, "Component Verification with Automatically Generated Assumption," Journal of Automated Software Engineering, Vol. 11, Kluwer, 2004.

[11] P. Gluck and G. Holzmann, "Using Spin Model Checking for Flight Software Verification," Proceeding of 2002 Aerospace Conference, March 2002.

[12] G. Holzmann, "Software Analysis and Model Checking," Proceedings of CAV 2002, July 2002.

[13] G. Holzmann, "The Spin Model Checker – Primer and Reference Manual," Addison-Wesley, September 2003.

[14] A. Jonsson, P. Morris, N. Muscettola, K. Rajan, and B. Smith, "Planning in Interplanetary Space: Theory and Practice." Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling, 2000.

[15] N. Muscettola and P. Nayak and B. Pell and B. Williams, "Remote Agent: To Boldly Go Where No AI System Has Gone Before." Artificial Intelligence,103(1-2), 1998.

[16] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin, Won Soo Kim, "CLARAty: An Architecture for Reusable Robotic Software," SPIE Aerosense Conference, April 2003.

[17] C. Pecheur and A. Cimatti, "Formal Verification of Diagnosability via Model Checking," Workshop on Model Checking and Artificial Intelligence, July 2002.

[18] R. Simmons and D. Apfelbaum, "A Task Description Language for Robot Control." Proc. of the Conference on Intelligent Robots and Systems (IROS), 1998.

[19] A. Venet, "A Scalable Nonuniform Pointer Analysis for Embedded Programs," Proceedings of SAS 04, Verona, Italy. LNCS 3148, Pp. 149-164, Springer 2004.

[20] A. Venet and G. Brat, "Precise and Efficient Static Array Bound Checking for Large Embedded C Programs," Proceedings of PLDI 2004, Washington, D.C., June 2004.

[21] J. Whittle and J. Schumann, "Automating the Implementation of Kalman Filter Algorithms," Accepted for publication in ACM Transactions on Mathematical Software (TOMS).