Asimov's Laws of Robotics Applied to Software

Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract

Asimov's Laws of Robotics constrain robots to server their human masters. Minor rewording shows that similar principles are very relevant to software too. These laws of software encompass a host of desiderata and tradeoffs that software developers need to keep in mind, and demonstrate that issues that are typically treated in a fragmented manner are actually strongly intertwined.

Introduction

In 1940, science fiction writer Isaac Asimov formulated the following three Laws of Robotics:

1. A robot may not injure a human being, or, through inaction, allow a human being to come to harm.
2. A robot must obey orders given it by human beings, except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

At the time, computers were in their infancy. Even Asimov did not foresee the prevalence of computers we see today, and the dominant role of software in governing how these general-purpose machines are used. Asimov therefore postulated that the Laws be implemented in hardware, by being embedded in the "positronic brains" of all robots.

Today it is clear that robots are and will be controlled by software. The high level of abstraction expressed by the three Laws therefore dictates that they would be implemented in the software controlling the robots. Indeed, the relevance and suitability of the Laws to robots and information technology in general have been discussed extensively in the context of controlling semi-autonomous systems that operate in the real world [3, 4, 13, 6]. However, it is not this possibility of directly implementing the three Laws that is the topic of this paper.

The original Laws of Robotics are focused on the physical well-being of the robot and its human masters. But they can also be applied to software in a more abstract manner, not directly

related to physical well-being. In this vein, we suggest the following laws of software. While they are largely a restatement of known principles, uniting them in this way serves to cast a new light on the contract between humanity and technology.

First Law

The first law is the most difficult to apply to software. If the software is not involved with the physical well-being of humans, how can it harm them? We therefore suggest that instead of focusing on the *physical* presence of humans in the real world, we focus on human presence *in cyberspace*.

The most direct manifestation of human presence in cyberspace is the execution of programs by computer users. Indeed, operating systems do not really know anything about human users — only about abstract users that run programs. In many cases the system can't even know if a program was executed by a human or by another program, because humans are represented by programs in any case. One might therefore postulate the first law for software as "Software may not harm the execution of programs by a human".

However, the important aspect of human computer usage is not the execution of a program *per se*; it is what the program does. For example, when I use a text editor to write this paper, the specific instance of a process running the text editor is much less important than the text I type in. Losing the process, while obviously undesirable, is nevertheless tolerable. Losing the text is much worse.

Based on such considerations, we can re-formulate the first Law as follows:

1. Software may not harm a human's work products, or, through inaction, allow the products of human work to come to harm.

This is in fact the statement of a basic requirement placed by Raskin on humane user interfaces, i.e. those that really try to serve their human users [11, p. 6]. (Raskin's precise formulation is slightly different, attributing the responsibility to the user interface rather to the application in general.)

While this formulation of the first Law is very concise, it has far-reaching implications. Some are naturally discussed in the context of the user interface, which is indeed the window to the application. Others relate to the internal workings of the software, with an eye to interoperability issues.

The first implication is the one emphasized by Raskin: user input is sacred, and it is intolerable to ever lose user input or data. He goes on to give some illuminating examples. One is the typing of text in a read-only window, or when no window has the focus. The system therefore does not know which application should receive the input [11, p. 175]. The simple way out (used by practically all current desktop systems) is to discard the input text, while possibly sounding an alarm bell to alert the user to the fact that the destination of the text is unknown. A better solution would be to pop up a special window that accumulates the typed text, and when the user subsequently assigns focus to a certain application, all the accumulated text is piped to that application and the popup window is discarded.

Another interface-related aspect of saving user data is the option to undo every command. Computer systems may not assume that the user is perfect and always knows what he is doing. On

the contrary, users may make mistakes. It is unacceptable that making such mistakes will lead to the irreversible loss of user data [11, p. 107].

At a somewhat more general level, retaining user data implies the support of automatic periodic saving of input to stable storage [11, p. 6]. This should not just be an option that can be activated by a knowledgeable user. It should be the default that prevents any user from losing data under any normal conditions. Taking such precautions meets the requirement that software not allow data to be lost through inaction.

Taking this to the next logical step, it is actually not enough to *store* the data. In addition, the data needs to be *accessible*. This has implications for the format in which the data is saved. Firstly, it is much better to store data in ASCII format (or possibly Unicode) than in some binary format. While this may inflate the volume of the data, this is not a real concern except for the largest of data sets (which are not generated manually by users). The benefit is that the data will likely be readable on different architectures for many years to come. Binary formats are architecture-specific, and may be hard to recover after some time has passed.

Secondly, it is preferable to use an open format rather than a proprietary one. When using a proprietary format, your data is at the mercy of the company that owns the format. Commercial interests may then prevent the creation of tools that support porting the data to systems from another vendor. This is especially problematic if the original vendor ceases to support the products used to store the data, e.g. if the vendor becomes insolvent. While such considerations are often overlooked, they are extremely important for the long term retainment of data in a world that is constantly changing [8, 2].

Thirdly, the data should be stored in a safe manner, protected from viruses and other malware that might corrupt it. These issues are discussed again below. In a related vein, user privacy should be protected. Thus the data should be stored in an accessible manner, but accessible only to the user, not to anyone. Reconciling long-term accessibility with restricting access by others seems to pose a non-trivial tradeoff.

An even wider interpretation of the First Law is that not only user data but also user *experience* should be protected. The word “experience” has two distinct meanings that are applicable here: experience as in knowledge about a procedure that is gained by repeatedly performing it, and experience as in the perception of how pleasant or easy it was to use the system.

The first meaning of experience is related to learning to use a software product. Today’s software products are often quite complex and packed with features. It takes a long time to discover all of them, what they do, and how to use them. This investment is often lost when the next version of the software is released, both because functionality may have changed, and because the same things may now look differently than they did before. The changes may seem small to the developers, who spend considerable time considering the best ways to perform various tasks. But they can be devastating to users who just want to perform their work with as little hassle as possible. As is often said, a good interface is one that disappears and is used with no apparent effort.

changing the interface makes is re-appear, and should only be done if the original interface is truly lacking, and never for cosmetic reasons. The impact on computer usability may be much larger than anticipated by the developers, and not necessarily for the better. The effect may be especially severe for the elderly, who were only introduced to computers at a relatively advanced

age. Children who learn to use computer applications by the same trial-and-error approach that helps them master computer games do not have difficulties with mastering new interfaces. I can sometimes help family members to perform some computer task not because I *know* how it is done, but because — having some experience in computer science and programming — I can enter the mindset of the software's developers and *guess* how it is done, or at least where to look. But people without the relevant background, and especially the elderly, may lack sufficient experience to figure it out.

The problem of changing interfaces is aggravated by the fact that interfaces tend to be bloated and counter-intuitive to begin with. Alan Cooper, in his book *The Inmates are Running the Asylum*, attributes this to design by engineers, favoring features over simplicity [5]. The result may overwhelm users who actually don't want so many features, and would prefer to focus on the simple and basic functionality.

The other type of experience that should be protected is the work experience. Productive work requires concentration, and it may take considerable time to "get into" the work at hand. Software should make every attempt not to interfere with this condition once it is attained.

For example, I sometimes hold down the shift key for some time when thinking exactly how to phrase the next sentence (which will start with a capital letter). In Windows, doing this for 8 seconds brings up a popup that explains about a feature called FilterKeys. Pressing the "cancel" button removes the popup, but sometimes also cancels the effect of the shift key, so when I finally decide what to write I find I cannot start the sentence with a capital letter. Restoring this takes some fiddling; by the time I get it to work, I typically have no idea what the sentence was supposed to be about. At the same time, the task bar contains an icon of two little computers with screens that flash on and off representing my wireless connection. A few minutes ago a popup appeared out of the blue announcing that new updates are available for my computer, and would I like to see them now. These are also distractions that make it harder to focus on the work at hand — writing this paper.

Second Law

The application of the Second Law to software may seem quite straightforward. In essence, it simply states that

2. Software must obey orders given it by its users.

These orders, of course, come from the repertoire of what the software is expected to do. In other words, this law simply states that software should function according to its specs. While the requirement that software do what it is supposed to do seems trivial, if not circular, there are actually some non-trivial implications. Two major ones are determining what the software should do in the first place, and who the user is.

Large and complex systems are notorious for their voluminous documentation, which is often hard to read and may also contain contradictions. Contrast this with typical user behavior, which

shuns reading any manuals, even short ones. Thus a broader interpretation of the Second Law is that software should be easy and intuitive to use — the holy grail of user interface design.

Moreover, software systems should have reasonable defaults and behave as expected even if nothing is said explicitly; they should do the right thing in a given context without this being spelled out (this may be the main major point missing in Asimov's original stories). In particular, many things are taken for granted and should just work as any reasonable user would expect. For example, when you type text you expect it to appear as you typed it, and this is indeed the case in the vast majority of cases. But some advanced word processors may modify your text, based on certain patterns, e.g. putting you in numbered list mode if a sentence starts with a numeral. Undoing this or turning off this behavior is typically much more involved than producing the original effect, and may be quite frustrating to novice users.

As another example, it is still very common today that text written in a combination of two languages with opposite directions (e.g. English and Hebrew) comes out garbled and requires extensive manual efforts to correct. Likewise, a system that requires a user to enter a date should be able to accept it in a wide variety of formats: 8/24, 08/24, 8.24, Aug 24, and August 24 are all the same thing, and humans can read them without even noticing which format was used.

But what users expect is actually context sensitive. Thus the date 8/1 would be read as August 1st in the US, but as the 8th of January in Europe — quite a significant difference. In software development terms, this implies an understanding of the context in which user actions are taken — not only context sensitive help, but also context sensitive operations. But note the important distinction between context and modes of operation. Modes are a part of the system state that causes the same user inputs to have different effects. This is bad because the user must remember what mode the system is in [11]. In contrast, context is gathered from user actions, and therefore is part of the user's mindset to begin with.

An even higher level of anticipation would be adaptive systems that learn common usage patterns and can repeat them. This is already a topic for AI and software agents. But in the context of simpler systems, we note that when such intelligent behavior is unattainable, the software should at least provide intelligent and informative error messages. User commands need not be a one-way stream, but can be fashioned as a dialog until mutual understanding of what has to be done is reached.

Related to the issue of executing commands is verifying that the user is allowed to issue them in the first place. Obviously, considerable advances have been made in security since the days when desktop systems simply assumed that anyone with access to the machine is allowed to do anything. But on the bottom line, this often boils down to passwords. The proliferation of passwords, coupled with requirements that they be hard to guess and be changed often, leads to situations in which users need to write them down to remember them — essentially relocating the vulnerability but not preventing it.

The situation is even worse in distributed and wireless systems. Maintaining security is indeed a constraint that may limit what can be done. Developers are notorious for preferring features over security, and in many cases relegating security to second class citizenship, only implemented as an afterthought. This is becoming increasingly unacceptable as the cost of breaches in security becomes prohibitive.

the issue of obeying orders can also be considered at a more basic level — that of reaching the state of being able to accept orders at all. In the context of software, this refers to the notorious issues of installation and configuration. The truth be told, much progress has been made in this area in recent years, with "installation wizards" that often automate the whole procedure. But this is typically limited to the vanilla, default case. Any deviation may lead to unexpected options and difficulties that are typically not explained in any reasonable manner.

The situation with open-source software is, if anything, even worse. Open source is typically developed by very knowledgeable people, who find it hard to imagine users who know much less than themselves. They therefore tend to fall into the trap of assuming the user has sufficient background to fill in the gaps. Moreover, testing is largely limited to the developer's environment, ignoring possible variations and their effect. An illuminating example is given by Eric Raymond, who recounts his experiences in trying to configure CUPS, the common Unix printing system; it required a few hours and large doses of experience from other systems that required similar setups. He therefore suggests that a safer approach for a developer is to imagine his aunt Tillie trying to install and configure the software, and to try not to leave her too far behind [12].

An interesting question is raised by the second clause in the original Second Law, which reads

> Except when such orders conflict with the First Law.

In principle, this wording applies equally well to software. It implies that software should resist causing damage. For example, if a virus infects a computer and instructs the system software to delete a user's files, the system software should resist and ignore these instructions. This is a very tall order, as it requires the system to distinguish between malicious and legal behavior that may look very similar. It would seem unreasonable to expect such judgment to be successfully administered by any software system. But at a more basic level, this is related to the security considerations discussed above.

Third Law

The software-oriented version of the Third Law is also quite straightforward:

> 3. Software must protect its own existence.

In other words, the software should be stable and should not crash. The fact that the user provided illegal input is no excuse.

In fact, most software is nowhere near as stable as we would like it to be. A striking demonstration of this situation is provided by Bart Miller's "fuzz" experiments [9]. In these experiments, various basic system utilities are given random inputs to see what will happen. In a distressingly large fraction of the cases, the utility crashes or hangs. Even worse, repeating the experiments five years later showed that many of the problems publicized in the first round had not yet been fixed [10].

At a deeper level, protecting itself means that software should also be robust against intended attacks. This includes two components. The first is resisting attacks by malware that attempt to

take over the system. The second is self-healing, i.e. being able to repair damages automatically — a property that has been called computer immunology [1].

Back in 1980, Tony Hoare stated in his Turing Award lecture that in any respectable branch of engineering bounds-checking would long have been required by law [7]. Twenty five years later, buffer overflows resulting from lack of bounds checking are a leading cause of security breaches. This reflects a carefree culture in which vendors and developers do not take responsibility for the results of their failures. And indeed, why should they if their customers continue to buy products based mainly on low perceived immediate price, and not on the potentially higher price should a failure occur. It seems that this culture will only change if and when a massive failure causes large losses to multiple users.

The original Third Law of Robotics included an additional clause — that the Third Law be followed provided it does not contradict one of the first two Laws. In software, it seems that the more common situation is that following the Third Law will be a special case of the first two laws. If a system allows itself to be taken over by malware, it exposes user data to harm, thus violating the First Law. If it actually causes harm, it is accepting orders from an unauthorized source, violating the Second Law. So protecting itself is actually a pre-requisite for protecting human work products and privacy in cyberspace, and serving the authentic human user.

Summary

In summary, we suggest that Asimov's Laws of Robotics can be interpreted in ways that are meaningful for general expectations from software systems. These interpretations include the following:

First Law: software should protect humans in cyberspace, including

- Never losing user data
- Protecting user privacy
- Storing user data in an open format in ASCII to allow it to be accessed by other software on future machines with different architectures
- Providing backward compatibility to protect user investment in learning how to use the software
- Not interrupting a user's interaction with an application

Second Law: software should obey commands, and in particular

- Be intuitive and easy to use
- Provide reasonable defaults that can substitute for explicit orders
- Provide informative error messages to guide users towards a solution
- Be easy to install and configure
- Protect against commands from unauthorized individuals

Third Law: software should protect itself, implying

- It should be stable and not crash
- It should be secure and resist viruses and other malware

Practically none of this is new; the laws of robotics simply provide a convenient framework to express all these desiderata and the inherent interactions and tradeoffs in a concise manner. On the other hand, much of this is also not common practice. The laws thus serve to focus attention on the fact that the software industry has been getting away with too much for too long. It is time for software developers to be more accountable for their products, and remember that their software is there to serve its users — just like Asimov's robots.

References

[1] M. Burgess, "*Computer immunology*". In 12th *Systems Admin. Conf. (LISA)*, pp. 283–297, USENIX, Dec 1998.

[2] S-S. Chen, "*The paradox of digital preservation*". *Computer* **34(3)**, pp. 24–28, Mar 2001.

[3] R. Clarke, "*Asimov's laws of robotics: implications for information technology, part 1*". *Computer* **26(12)**, pp. 53–61, Dec 1993.

[4] R. Clarke, "*Asimov's laws of robotics: implications for information technology, part 2*". *Computer* **27(1)**, pp. 57–66, Jan 1994.

[5] A. Cooper, *The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity*. SAMS, 2nd ed., 2004.

[6] D. F. Gordon, "*Asimovian adaptive agents*". *J. Artificial Intelligence Res.* **13**, pp. 95–153, 2000.

[7] C. A. R. Hoare, "*The emperor's old clothes*". *Comm. ACM* **24(2)**, pp. 75–83, Feb 1981.

[8] K-D. Lehmann, "*Making the transitory permanent: the intellectual heritage in a digitized world of knowledge*". In *Books, Bricks & Bytes: Libraries in the Twenty-First Century*, S. R. Graubard and P. LeClerc (eds.), pp. 307–329, Transaction Publishers, New Brunswick, NJ, 1999.

[9] B. P. Miller, L. Fredriksen, and B. So, "*An empirical study of the reliability of UNIX utilities*". *Comm. ACM* **33(12)**, pp. 32–44, Dec 1990.

[10] B. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*. Technical Report, University of Wisconsin — Madison, 1995.

[11] J. Raskin, *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley, 2000.

[12] E. Raymond, "*The luxury of ignorance: an open-source horror story*". URL http://www.catb.org/~esr/writings/cups-horror.html, Jul 2004.

[13] D. Weld and O. Etzioni, "*The first law of robotics (a call to arms)*". In 12th *Natl. Conf. Artificial Intelligence (AAAI)*, vol. 2, pp. 1042–1047, Jul 1994.