# Towards Fixing Sketchy UML Models by Leveraging Textual Notations: Application to Real-Time Embedded Systems

Frédéric Jouault and Jérôme Delatour

LUNAM, L'Université Nantes Angers Le Mans
TRAME team, ESEO, Angers, France
firstname.lastname@eseo.fr

**Abstract.** During the development of real-time embedded system, the use of UML models as blueprints is a common practice with a focus on understandability rather than comprehensiveness. However, further in the development, these models must be completed in order to achieve the necessary validation and verification activities. They are typically performed by using several formal tools. Each tool needs models of the system at a given abstraction level, which are precise and complete enough with respect to how the tool processes them. If UML is appropriate for capturing multiple concerns, its multiple partial views without a global one increase the difficulty of locating inconsistency or incompleteness issues. Therefore, ensuring completeness is time consuming, fastidious, and error prone. We propose an approach based on the use of a UML textual syntax closely aligned to its metamodel: tUML. An initial prototype is described, and examples are given.

## 1 Introduction

Using sketchy models is common during software systems development. Such models (also called contemplative models) are just drawings that cannot be automatically processed (e.g., transformed into code, or verified). In the use of such models as blueprints, the focus is on communication rather than on completeness. The UML language is a good candidate for that kind of practice. It is primarily a graphical notation, and it offers a relatively large set of diagrams for representing various concerns. Moreover, as it is a standard, one can expect that it is known by developers, and also by stakeholders. Therefore, it helps communicating a better understanding about the system under study, and supports a better collaboration with various specialists.

However, such contemplative use of UML models is only partially satisfactory. Part of the effort to produce them cannot be automatically reused in further development phases. Going from contemplative to productive models is a key challenge of model-driven approaches [4]. In these approaches, development can be seen as a set of successive model transformations steps in order to produce code from analysis models. Each step should be partially automated.

In real-time embedded system development, this is even more important. The reliability and complexity of such systems require numerous verifications and validations activities all along the development process. These activities are based on the use of tools such as model checkers, theorem provers, code generators, and test generators. The use of UML in this context presents some issues. It has no standard formal semantics, and many semantic variation points. A many-diagram model may contain inconsistencies, and no global view helps their discovery. UML is semi-formal: a given drawing may have different interpretations. UML expressivity is not completely satisfactory for this domain.

Solutions to these issues have been developed. UML extensions have been defined (e.g., the MARTE[1] profile) to adapt it to the real-time domain. Numerous works propose a variety of translations from part of UML to different formal languages [6,10,9]. These translations enable detection of some inconsistencies [5], and other verification activities.

These solutions require that the verified model parts be precise, complete, and unambiguous. Other parts may remain sketchy [12]. However, it is difficult for an engineer to go from a sketchy model to a precise one. This typically requires detailed knowledge of: the UML specification, and the specific semantics given to it by a given verification tool [2]. Even if some UML editors assist users, this work is difficult and long. It is generally necessary to fill-in hundreds of property sheets accessible from a potentially large number of distinct graphical views.

In this work, we propose an approach to bridge the gap between sketchy and (partially) precise models. We put some efforts in making it as independent of specific UML editors as possible. It is based on the use of a specific textual syntax called tUML that closely follows the UML metamodel structure, as well as on the use of additional constraints on models. Neither text nor graphical is better than the other. However, by complementing graphical tools with textual ones, we can get both their respective advantages.

We illustrate our approach with the pacemaker case study presented in [3]. It is actually while working on this case study that we initially had the idea to develop the presented approach. We started by playing the role of the designer defining a sketchy model. Then, we had to complete it so that it can be verified. This task is now much easier thanks to our tUML prototype.

Section 2 presents the approach in four steps: an overview of tUML (Section 2.1), a description of our prototype (Section 2.2), an explanation of its usage (Section 2.3), and a discussion (Section 2.4). Then, Section 3 compares tUML to some related works. Finally, we conclude in Section 4.

## 2 Approach

### 2.1 tUML Presentation

tUML is a textual concrete syntax for a subset of the standard UML metamodel. A detailed definition and justification of this subset is beyond the scope of this

[1] `http://www.omgmarte.org/`

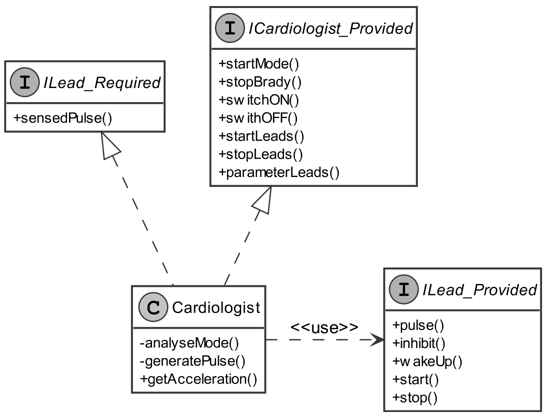


**Fig. 1.** Class diagram of `Cardiologist`

paper. Roughly, it consists of the parts of the UML metamodel that correspond to the three following diagrams: 1) class diagrams, 2) composite structure diagrams, and 3) state diagram. These diagrams are typically the ones used to model real-time embedded systems. We do not attempt to cover the whole UML metamodel, but built our subset bottom-up, making sure we can verify what we model [8]. Additionally, we use a specific action language restricted to what we can verify.

In order to illustrate: 1) the tUML textual syntax, and 2) its relations with the standard graphical notation, we use the pacemaker case study introduced in [3]. In this case study, a pacemaker is designed as a UML model, of which a full description does not belong here. Therefore, we only briefly mention what is necessary to understand this paper. A `Cardiologist` active object is responsible for applying a heart-assistance policy such as `TriggeredPacing` by piloting `Leads`. A `Lead` active object brokers access to actual physical leads placed on the heart of the patient. `Cardiologist` and `Lead` are software components that are part of an implanted `PulseGenerator`. The overall `System` consists of a `PulseGenerator`, which can communicate with an external `DeviceControlMonitor`. We reuse three figures from [3], which correspond to our three UML metamodel parts of interest: classes, composite structure, and state machines.

**Classes.** Figure 1 (corresponding to Figure 9 from [3]) is a partial view of class `Cardiologist`, and related interfaces in the standard class diagram notation. `Cardiologist` implements interface `ICardiologist_Provided`[2], which defines messages it can receive from its controller. It also implements interface `ILead_Required`, which defines messages it can receive from `Leads`. `Cardiologist` uses interface `ILead_Provided`, which defines messages it can send to `Leads`.

Listing 1 is an excerpt of the corresponding tUML definition. Class `Cardiologist` is defined as implementing two interfaces (line 1), and using a third one

[2] All interface names from original case study have been simplified to increase clarity.

**Listing 1.** `Cardiologist` excerpt

```
1  class Cardiologist behavesAs SMCardiologist implements
       ↪ICardiologist_Provided , ILead_Required {
2    // [ports]
3    private operation analyseMode();
4    private operation generatePulse();
5    public operation getAcceleration();
6    stateMachine SMCardiologist {
7      region Region0 {
8        // [state machine contents]
9  }}}
10 usage Usage0 of ILead_Provided by Cardiologist;
```

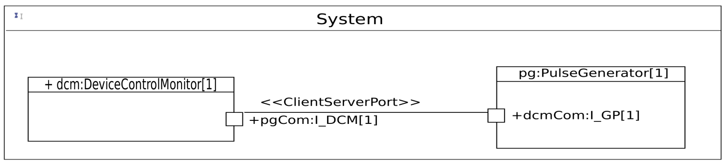


**Fig. 2.** Composite Structure diagram of `System`

(line 10). These interfaces have the same names as those in Figure 1. Lines 3 to 5 contain the definitions of operations owned by the class. Comments starting with two slashes end with the line. They are used here to stand for actual ports (line 2) and state machine contents (line 8), which are not detailed. Lines 6 to 9 correspond to an element not shown in the class diagram: a state machine (not detailed here) that specifies the behavior of this active object.

**Composite Structure.** Figure 2 (Figure 5 in [3]) is a partial view of the `System`'s architecture in the standard composite structure diagram notation. This architecture consists of a `DeviceControlMonitor` named `dcm` owning a port called `pgCom`, a `PulseGenerator` named `pg` owning a port called `dcmCom`, and a connector between the ports.

Listing 2 is an excerpt of the corresponding tUML definition. Class `System` is defined at lines 1-4 as consisting of a `PulseGenerator` part at line 2, a `Device-ControlMonitor` part at line 3, and a connector at line 4. Class `PulseGenerator` is defined at lines 5-9 with a port at line 7. This class is also composite but its parts (line 6) and connectors (line 8) are not detailed here. Class `DeviceCon-trolMonitor` is defined at lines 10-11, and only contains a port (line 11).

**State Machines.** Figure 3 (Figure 12 in [3]) is a partial view of a composite substate of `Cardiologist`'s behavioral state machine in the standard state diagram notation. State `TriggeredPacing` is composed of states `StartingPulse` and `WaitingPulse`, as well as of an initial pseudo-state. It also contains four transitions: 1) one transition from the initial pseudo-state to `StartingPulse`, 2) one transition from `StartingPulse` to `WaitingPulse` triggered after a given

**Listing 2.** `System` composite structure

```
class System {
  public composite pg[1-1] unique : PulseGenerator;
  public composite dcm[1-1] unique : DeviceControlMonitor;
  connector Connector0 between dcm.pgCom and pg.dcmCom; }
class PulseGenerator {
  // [parts]
  public composite port dcmCom[1-1];
  // [connectors between parts]
}
class DeviceControlMonitor {
  public composite port pgCom[1-1]; }
```

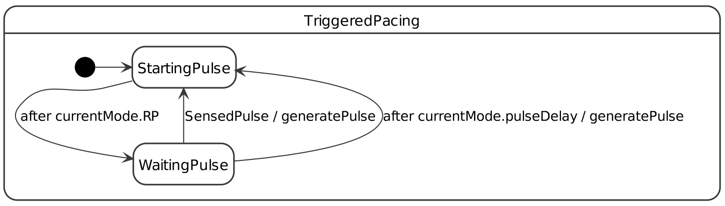

TriggeredPacing
StartingPulse
after currentMode.RP
SensedPulse / generatePulse
after currentMode.pulseDelay / generatePulse
WaitingPulse

**Fig. 3.** State diagram showing part of the behavior of `Cardiologist`

time (`currentMode.RP`), and two transitions from `WaitingPulse` to `StartingPulse` triggered: 3) by `SensedPulse` message, with `generatePulse` effect, and 4) after a given time (`currentMode.pulseDelay`), with the same effect.

Listing 3 is an excerpt of the corresponding tUML definition. State `TriggeredPacing` is defined at lines 1-9 as consisting of region `Region0`. This region itself consist of an initial pseudo-state (line 7), two states `StartingPulse` (lines 8) & `WaitingPulse` (line 9), and four transitions (lines 3-6). Each transition has a name. Each transition with a trigger refers to an event by name (escaped between double quotes because they contain spaces). Events are defined in a part of the model not shown here. Finally, the two transitions having an effect specify it as an empty opaque behavior (lines 5-6), with only its name specified.

## 2.2 Prototype

Now that tUML has been presented, we will describe its current prototype implementation. Eclipse Modeling[3] has been used in the following way:

- **Eclipse UML** provides the UML abstract syntax implementation (very close to the standard), on top of EMF (Eclipse Modeling Framework). Tools based on Eclipse UML (e.g., Papyrus) are directly compatible with tUML.
- **TCS** [7] is used to define the textual syntax. It provides three elements: 1) an *extractor* from Eclipse UML abstract syntax to concrete tUML textual syntax, 2) an *injector* from tUML textual syntax to Eclipse UML, and 3) an *editor* (see Figure 4) tuned for tUML textual syntax.

[3] `http://www.eclipse.org/modeling/`

**Listing 3.** `Cardiologist` state machine except

```
state TriggeredPacing {
  region Region0 {
    Initial0->StartingPulse : Transition0;
    StartingPulse->WaitingPulse : Transition1 : "TE - currentMode.RP" /;
    WaitingPulse->StartingPulse : Transition2 : "SE - SensedPulse" /
        ↪opaqueBehavior generatePulse;;
    WaitingPulse->StartingPulse : Transition3 : "TE - currentMode.
        ↪pulseDelay" / opaqueBehavior generatePulse;;
    initial pseudoState Initial0;
    state StartingPulse;
    state WaitingPulse;   }}
```

– **ATL** [1] transformations are used to implement four elements: 1) *constraints* (in OCL) to check on tUML models, 2) *qualified names* computation and resolution whereas TCS only supports simple names by itself, 3) *visualization* in the form of a PlantUML[4] export, and 4) *bridges* with other tools.

Other tools could have been used (e.g., Xtext), but we prefer familiar ones.

The PlantUML export enables graphical rendering of tUML models. Figures 1 and 3 have actually been generated through this process. Figure 2 is however still the original from [3] because PlantUML cannot properly render it yet. A non-standard context diagram can also be rendered. It is a specialization of the communication diagram in which a non-ordered list of all exchanged messages (with directions) is shown on each edge.

Our prototype notably has the following limitations. Conversions between textual and abstract syntax currently work on whole models, and graphical information is not synchronized yet. Fine-grain synchronization would be helpful but is not mandatory in our approach. Apart from tools based on Eclipse UML, only a Rhapsody import has been implemented so far.

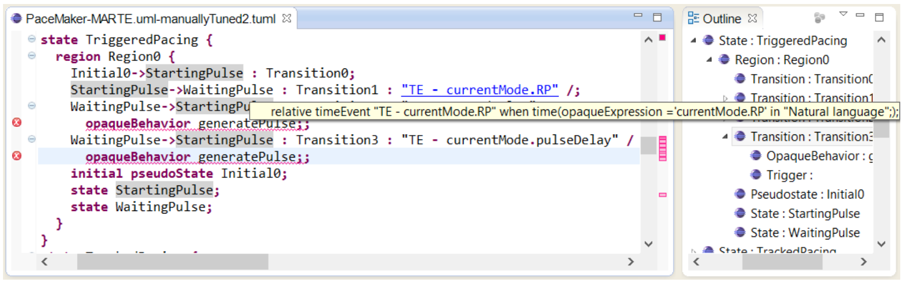
### 2.3 Using tUML to Fix Sketchy Models

The previous sections gave an overview of tUML and of its prototype implementation. This section will give an overview of how they can be used to fix sketchy models. Before creating models and fixing them, it is first necessary to consider how they will be used (e.g., for code generation, or verification). Then, an expert can define what a *correct* model precisely is in that context. This will typically lead to a list of custom validation constraints. This work does not need to be performed for each model, but only for each usage of models (e.g., for a given code generator, or for a given verification tool).

The user typically follows the five-step process described below:

1. **Creating Sketchy Model.** A modeling tool is first used to create a sketchy model. This is generally done with a graphical UML editor.
2. **Converting to tUML.** Using the extraction tool presented previously, the user serializes the model in tUML.

[4] `http://plantuml.com/`



```
state TriggeredPacing {
  region Region0 {
    Initial0->StartingPulse : Transition0;
    StartingPulse->WaitingPulse : Transition1 : "TE - currentMode.RP" /;
    WaitingPulse->StartingPu
      opaqueBehavior generatePulse;;
    WaitingPulse->StartingPulse : Transition3 : "TE - currentMode.pulseDelay" /
      opaqueBehavior generatePulse;;
    initial pseudoState Initial0;
    state StartingPulse;
    state WaitingPulse;
  }
}
```

```
relative timeEvent "TE - currentMode.RP" when time(opaqueExpression ='currentMode.RP' in "Natural language";);
```

**Fig. 4.** Screenshot of tUML editor showing composite state `TriggeredPacing`

3. **Identifying Problems.** Issues are identified by applying standard UML constraints, as well as custom constraints if applicable. They are shown in the tUML editor. If no problem is identified, then we can go to step 5
4. **Fixing Problems.** The user can tune the model in order to fix problems. This is often easier to do textually because all information is presented at once. In comparison, a graphical editor often requires opening various property sheets. Then, we go back to step 3 in order to check if the tackled issues are fixed, and if some issues remain.
5. **Use Fixed Model.** Now that all identified problems have been fixed, the model can be used in four notable ways: 1) **reloading** the model in the original UML tool, or another one, 2) **generating** code, 3) **verifying** the model using formal methods, and 4) **rendering** the model using tools such as PlantUML (see prototype description).

Our pacemaker example was initially created as a sketchy model, not to be automatically processed. Later, we decided to attempt to use some verification tools [8]. We defined five specific constraints in addition to the standard ones. tUML helped us discover and pinpoint (see error markers in Figure 4) more than thirty issues mostly in the modeling of communication. We can notably mention: 1) opaque behaviors with no body (e.g., in Listing 3, which is not obvious on Figure 3), 2) undetailed communications (e.g., call to private operation `generatePulse` with no body), and 3) use of non-defined signals in triggers. To fix them, we select a language for opaque behaviors, and we fill-in the blanks.

## 2.4 Discussion

After presenting tUML, its implementation, and its usage, we now discuss the advantages and drawbacks of our approach. Firstly, tUML has all the benefits of a textual language, such as: 1) global search and replace, 2) relatively easy editing with no need to click through numerous property pages, 3) support for diff/patch and text-based version control systems, and 4) no need to focus on graphical layout (indentation being much easier). Other advantages of tUML when compared with the standard UML graphical notation include:

- **Transparency.** All model elements are immediately visible, and no aspect of the model is hidden away in property sheets accessible via multiple clicks.
- **Global view.** Whereas the graphical notation fragments the model into separate diagrams, tUML provides a single global view.
- **Reduced redundancy.** There is no need to duplicate elements (even partially) across various diagrams.
- **Proximity to metamodel.** The textual syntax of tUML follows the UML metamodel very closely.

Proximity to the UML metamodel notably has the advantage that it is relatively easy to figure out where a given aspect of the UML metamodel can be accessed. Moreover, this enables relatively precise reporting of errors as text markers, even though they are actually identified on the abstract syntax. Graphical editors cannot always do this. For instance, in Papyrus, some errors on class members are shown on their owning class in the class diagram.

Although this is not due to the textual nature of tUML, the automatic generation of views presents some advantages as well. Even though tUML models typically come from graphical UML editors, this has the advantage of reformulating the models in a different way, possibly helping to spot issues. Moreover, the non-standard context diagram is especially helpful in finding messages that are missing (e.g., when their emission or reception are not properly modeled).

Of course, tUML also has some limitations:

- **Verbose.** All model elements have to appear in a single text file. Therefore, even though they are visible, details of interest may actually be drowned in unnecessary details. Global view is thus both an advantage and a drawback. To mitigate this issue, generation of partial textual views should be investigated. Synchronization with the whole model may become an issue.
- **Non-standard.** Because it is not standard, tUML needs to be learned even by UML experts. This adds one difficulty to its usage.
- **Incomplete.** Only a subset of UML is currently supported. This subset is especially targeted at real-time embedded systems. Using tUML in other contexts may prove difficult (e.g., if one needs activities).
- **Textual.** Finally, even though most advantages of tUML are due to its textual nature, some users may not accept to type UML code.

## 3 Related Work

UML is best known for its graphical notation, but tUML is not the first proposed UML textual notation. The OMG has notably standardized two of its ancestors[5]:

- **HUTN** initially seems to provide a solution since it promises automatic derivation of a human-usable textual notation from any metamodel. However, this genericity leads to relatively verbose syntaxes.

[5] `http://www.omg.org/spec/HUTN/`, and `http://www.omg.org/spec/ALF/`

- **Alf** defines a specific textual syntax for a UML subset. The OMG thus acknowledges the fact that HUTN is not fully adapted to this kind of use. Unfortunately, Alf is limited to a different subset of UML, which is not adapted to real-time embedded systems. tUML is thus significantly different from Alf. Notably, Alf only provides activities to specify behavior, whereas tUML focuses on state machines. Moreover, Alf specifies a new metamodel that is closer to its concrete syntax than the standard UML metamodel. Although a bidirectional mapping to the standard UML metamodel is also provided, such a gap between textual and graphical notations may make the job of fixing models for formal verification purposes more difficult.

Most non-standard textual UML notations aim at producing sketchy models by leveraging auto-layout tools. We cannot describe them all[6] here, but two notable ones are:

- **TextUML**[7] is relatively close to tUML but does not follow the UML metamodel as closely (notably wrt. its action language).
- **PlantUML** has the advantage of being less verbose and more permissive than all other mentioned approaches. However, its objective is only to automatically generate visual diagrams. It does not follow the standard metamodel. Moreover, it is so permissive that many elements (e.g., class members, and transition labels) can be represented by arbitrary text. Apart from following very strict conventions, there is no way to guarantee that these elements actually follow the UML syntax.

Some approaches like UML/P [11] also offer a textual representation of UML models, and specific constraints to check them. However, the supported UML subset differs: UML/P is not specific to real-time embedded systems, and notably includes additional diagrams that our approach [8] is not yet able to verify. It also seems to lack composite structure diagram, which we need.

## 4 Conclusion

In this paper, we have presented an approach that helps fixing incomplete UML models to make them suitable as input for verification and validation activities and tools. This approach relies on tUML: a specific textual notation for UML that notably follows its metamodel very closely. A prototype implementation has been used on a couple of case studies (including the pacemaker one presented here), and half a dozen different models. tUML significantly helped in discovering issues, and therefore in fixing them.

We have so far demonstrated the worth of the approach, but its development is still in progress. It should be further studied and refined, notably by applying it to more case studies, including industrial ones. This would lead to a complete evaluation. This should bring out more kinds of constraints to check on models,

[6] See `http://modeling-languages.com/uml-tools/#textual` for a bigger list.

[7] `https://github.com/abstratt/textuml/`

as well as help figuring out in which direction the supported UML subset should be extended. All sketchy models issues cannot be resolved automatically. It could nonetheless be useful to provide quick fixes[8] in order to not only pinpoint issues, but also partially automate their resolution.

A possible extensions of our work on tUML would be to propose it to the OMG as an extension of Alf, which notably lacks support for state machines.

## References

1. Bézivin, J., Jouault, F.: Using ATL for Checking Models. Electronic Notes in Theoretical Computer Science 152, 69–81 (Mar 2006), proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)
2. Broy, M., Cengarle, M.: UML formal semantics: lessons learned. Software & Systems Modeling 10(4), 441–446 (2011)
3. Delatour, J., Champeau, J.: Embedded Systems: Analysis and Modeling with SysML, UML and AADL, chap. Case Study Modeling using MARTE, pp. 139–156. Wiley-ISTE, West Sussex, England (Apr 2013)
4. Gérard, S., Feiler, P., Rolland, J.F., Filali, M., Reiser, M.O., Delanote, D., Berbers, Y., Pautet, L., Perseil, I.: UML & AADL '2007 Grand Challenges. SIGBED Rev. 4(4) (Oct 2007)
5. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) Tests and Proofs, Lecture Notes in Computer Science, vol. 5668, pp. 90–104. Springer Berlin Heidelberg (2009)
6. Grönniger, H., Rumpe, B.: Considerations and Rationale for a UML System Model. UML 2 Semantics and Applications p. 43 (2009)
7. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the 5th international conference on Generative programming and component engineering. pp. 249–254. ACM (2006)
8. Jouault, F., Teodorov, C., Delatour, J., Le Roux, L., Dhaussy, P.: Transformation de modèles UML vers Fiacre, via les langages intermédiaires tUML et ABCD. Génie logiciel 109, 21–27 (Jun 2014)
9. Liu, S., Liu, Y., André, É., Choppy, C., Sun, J., Wadhwa, B., Dong, J.: A Formal Semantics for Complete UML State Machines with Communications. In: Johnsen, E., Petre, L. (eds.) Integrated Formal Methods, Lecture Notes in Computer Science, vol. 7940, pp. 331–346. Springer Berlin Heidelberg (2013)
10. Lund, M., Refsdal, A., Stølen, K.: 4 Semantics of UML Models for Dynamic Behavior. In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.) Model-Based Engineering of Embedded Real-Time Systems, Lecture Notes in Computer Science, vol. 6100, pp. 77–103. Springer Berlin Heidelberg (2010)
11. Rumpe, B.: Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring. Springer Berlin, 2nd edn. (Jun 2012)
12. Selic, B.: The pragmatics of model-driven development. IEEE Softw. 20(5), 19–25 (Sep 2003)

[8] A technique coming from programming environments (see `http://wiki.eclipse.org/FAQ_What_is_a_Quick_Fix%3F`).