Trustworthiness Derivation Tree: A Model of Evidence-Based Software Trustworthiness

Yuxin Deng*, Zezhong Chen*, Wenjie Du†, Bifei Mao‡, Zhizhang Liang‡, Qiushi Lin‡, and Jinghui Li‡
*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
†Shanghai Normal University, Shanghai, China
‡Trustworthiness Theory Research Center, Huawei Technologies Co., Ltd., Shenzhen, China

Abstract—**In order to analyze the trustworthiness of complex software systems, we propose a model of evidence-based software trustworthiness called trustworthiness derivation tree (TDT). The basic idea of constructing a TDT is to refine main properties into key ingredients and continue the refinement until basic facts such as evidences are reached. The skeleton of a TDT can be specified by a set of rules, which is convenient for automated reasoning in Prolog. We develop a visualization tool that can construct the skeleton of a TDT by taking the rules as input, and allow a user to edit the TDT in a graphical user interface. In a software development life cycle, TDTs can serve as a communication means for different stakeholders to agree on the properties about a system in the requirement analysis phase, and they can be used for deductive reasoning so as to verify whether the system achieves trustworthiness in the product validation phase. We have piloted the approach of using TDTs in more than a dozen real scenarios of software development. Indeed, using TDTs helped us to discover and then resolve some subtle problems.**

Index Terms—**Trustworthiness, refinement, evidence, visualization**

I. Introduction

If we are given a complex software or a system that combines both software and hardware, it is natural to ask if the system is trustworthy. Especially with the development of artificial intelligence technology and the widespread applications of unmanned vehicles, robots and other equipment, the issue of system trustworthiness is attracting more and more attention.

Trustworthiness is a very complicated property that involves the whole life cycle of a system and depends on a great number of factors. Firstly, the life cycle of a complex system goes through several stages such as requirement analysis, system development, delivery and maintenance, where every stage accounts for the trustworthiness of the whole system. Secondly, as a holistic property, system trustworthiness itself is too abstract to be verified. We need to refine it into different components or sub-properties. For example, it is identified in [10] to predominantly consist of the following six sub-properties: safety, reliability, availability, privacy, resilience and security. When the refinement is reasonable and each sub-property is validated, we can then derive that the main property of system trustworthiness is validated. Lastly, there are different ways of argumentations for the rationale of refinement and the validity of sub-properties. Some are informal inferences by using natural languages, while others are formal reasoning based on mathematical logics. Generally speaking, reasoning with formal methods requires the users to be well trained on mathematical logics. However, due to their rigidity and the possibility of being supported by automated or semi-automated tools, formal methods are playing an increasingly important role in the area of software engineering.

In the current work, we propose an evidence-based model of system trustworthiness, called *trustworthiness derivation tree* (TDT), which will be useful for communication and knowledge management of the analysis of system trustworthiness. The basic idea of constructing a TDT is to refine main properties into key ingredients and continue the refinement until basic facts such as evidences are reached. By following the whole process of refinement, we obtain a finite tree. As long as the refinement is sound, then the property corresponding to a node is valid if the sub-properties associated with all its child nodes are valid. In this way, if all the properties at the leaves of the tree are verified, then the trustworthiness at the root node holds. We will use formulas in first-order (predicate) logic to specify properties, and use a rule to specify the implication relation between a node and its children, which can be accepted by Prolog [14]. Therefore, we can take advantage of the inference capability of Prolog to check if the property associated with a node in a TDT is sound.

As a general framework of argumentation, the model of TDTs allows us to reason as formally as necessary about all the properties we are interested in. In the requirement analysis phase of software development, it can serve as a communication means for different stakeholders to agree on the properties about a system. It does not restrict the way of argumentations for the soundness of each rule, which could be formal or informal methods, and thus increase the flexibility of the framework. For example, the model of *assurance cases* (sometimes called *safety cases*) [4], [5], [8] can be embedded in our framework. In the product validation phase, TDTs can be used for deductive reasoning so as to verify whether the system achieves trustworthiness.

If a TDT has a great many nodes, it is inconvenient to manually construct, manipulate and maintain the tree. Motivated by this observation, we design and develop a visualization tool for TDTs. A user can specify the skeleton of a TDT in a text file as a set of Prolog rules. By taking that file as input, the tool is able to render a tree in a graphical user interface. A user can interactively edit the tree such as adding and deleting nodes, modifying the content of a node etc. One can also drag nodes as in UML tools, hide or display the subtree under a node, rotate the whole tree, zoom in or out etc. The skeleton of the tree can be exported as Prolog rules. The whole tree can also be saved in a CSV file, which is convenient if a team collaborates to construct and maintain a large TDT.

The rest of the paper is structured as follows. In Section II we discuss the issue of trustworthiness assurance. In Section III we introduce the model of trustworthiness derivation trees. In Section IV we introduce the visualization tool we have developed for TDTs. In Section V we discuss the use of TDTs in two phases of a software development life cycle. Finally, we conclude in Section VI.

II. Trustworthiness assurance

Closely related to trustworthiness is the concept of trust. There are various definitions of trust in different disciplines, and a careful analysis of them shows that [10]:

- Trust and trustworthiness form an asymmetric bidirectional relation. Trust is the attitude of a trustor, e.g. a customer, towards whether its trustee, e.g. a supplier, is trustworthy in an uncertain situation, while trustworthiness is the trustee's attitude towards itself and its products in an uncertain situation.
- Trust and trustworthiness are extensively context-sensitive. Relevant impacting factors need to be identified comprehensively via an analysis of the context.

When a system brings business value to its customers in a defined context, it also needs to provide the capability to handle adverse events and manage potential risks properly. Therefore, the trustworthiness of a system depends on its capability to manage potential risks, which involves three key parts: the risks faced by the system, the mitigation measures to address the risks, and the verification of the measures. The assurance of trustworthiness is then the argumentation about whether the mitigation measures for the risks are correct and sufficient, whether the mitigation measures are correctly implemented and whether the verification methods could tell the differentiation degree.

In safety engineering, the model of *assurance cases* [8] can be used for arguing for the safety of a product. It finds application in energy, aviation, aerospace, railway, automobile, medical and many other safety-critical areas [11], [12]. An assurance case is a documented body of evidence that provides a valid argument so that a specified set of claims regarding a product's properties are adequately justified for a given application in a given environment. Assurance cases can be presented graphically in Goal Structuring Notation (GSN) [9] or Claims-Argument-Evidence (CAE) [6]. A simple example of CAE fragment with side-warrant [7] is shown in Figure 1, which includes the following ingredients:

- A claim is an assertion about a product or a system. The top-level claim is the conclusion or final claim, and sub-claims are used to support higher-level claims. A sub-claim can either be further decomposed or be witnessed by an evidence, which can be a fact, a collection of data, or a physical object etc.

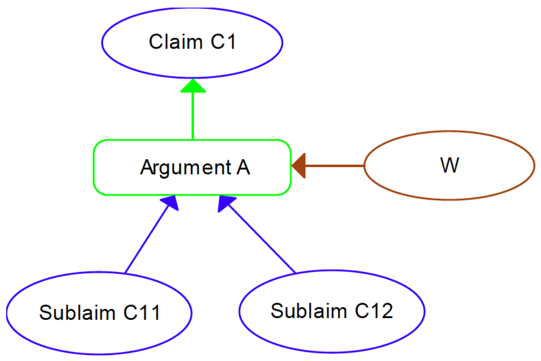


Fig. 1: An example of CAE fragment

- An argument is a reasoning method to prove that a claim is true.
- A side-warrant is a justification to the argument. It gives the reasons of deducing the top-level claim from the sub-claims and under what circumstances the argument is valid.

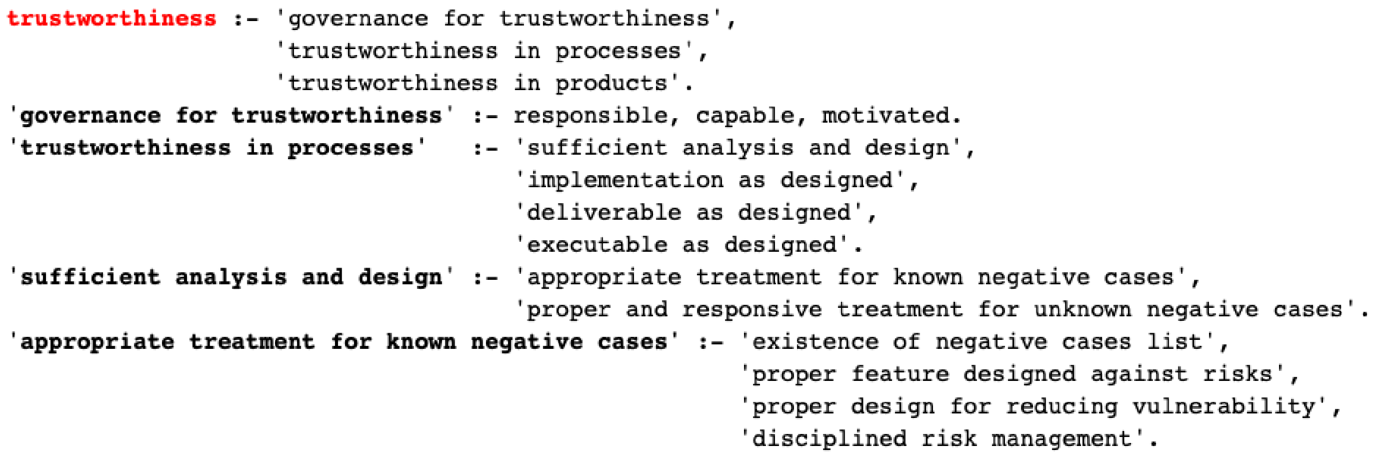
We can see that the sub-claims are used to support the claim at the top. The argument and side-warrant are auxiliary ingredients as they help to justify the refinement of the claim. In the current work, we propose to simplify the model of assurance cases by considering the auxiliary ingredients as a description of the claim, while keeping the key part of claim refinement. The resulting model is called trustworthiness derivation tree (TDT). In other words, we focus on the skeletons of assurance cases without loosing their expressiveness, as an assurance case can be converted into a TDT by writing its arguments and side-warrants as descriptions of claims. TDTs will be useful for communication and knowledge management of the analysis of system trustworthiness.

As a general framework of argumentation, the model of TDTs allows us to reason as formally as necessary about all the properties we are interested in. Since trustworthiness is context-sensitive, the meaning of trustworthiness is not widely agreed, and needs to be determined through dialogues between different stakeholders. Syntactically, a TDT takes the shape of a finite tree for property refinement. Semantically, the relationship between each parent node and its child nodes in a TDT represents an inference, a dependency, or an implication. That is where the name of TDT comes from.

In terms of tool support, a number of tools [1] implement the GSN notation, and Adelard ASCE tool implements the CAE notation. They assist in reading and reviewing assurance cases. However, none of them supports automated reasoning. For the model of TDTs, on the contrary, it is easy to perform rule-based reasoning, as we will see in the next section.

III. Trustworthiness derivation trees

We first introduce the concept of trustworthiness derivation tree, present a method of constructing those trees, and propose a way of automated verification of the trees.



```
trustworthiness :- 'governance for trustworthiness',
                   'trustworthiness in processes',
                   'trustworthiness in products'.
'governance for trustworthiness' :- responsible, capable, motivated.
'trustworthiness in processes'   :- 'sufficient analysis and design',
                                    'implementation as designed',
                                    'deliverable as designed',
                                    'executable as designed'.
'sufficient analysis and design' :- 'appropriate treatment for known negative cases',
                                    'proper and responsive treatment for unknown negative cases'.
'appropriate treatment for known negative cases' :- 'existence of negative cases list',
                                                    'proper feature designed against risks',
                                                    'proper design for reducing vulnerability',
                                                    'disciplined risk management'.
```

Fig. 2: A set of rules

Definition 1: A *trustworthiness derivation tree (TDT)* is a finite tree with each node labelled by a pair $\langle P, D\rangle$, where P is a formula in first-order logic representing a property of trustworthiness and D is a description about the property.

If we only care about the verification of the properties of trustworthiness, then the description of the properties can be omitted. Such a simplified form of TDT is called the *skeleton* of the TDT. Below we consider the problem of generating a skeleton from a set of rules.

Definition 2: A *rule* is a tuple of formulas in the form $\langle P_1, ..., P_n, P\rangle$, where each P_i is a *premise* of the rule, and P is the *conclusion* of the rule. If $n = 0$, that is, the rule has no premise, then the special rule is called an *axiom*.
In the current work, we use the syntax of Prolog to write rules and axioms. For example, we write "$P\ :-\ P_1, P_2, ..., P_n$." for a rule with conclusion P and premises $P_1, ..., P_n$, and write "P." for the axiom P.

Let T be a TDT such that the refinement of each node into its children is unambiguous. In other words, if two nodes are associated with the same property, then their children must be exactly the same. It follows that we have a set of rules that can generate T. More specifically, since T is a finite tree, each non-leaf node labelled by A has finitely many children with properties, say $A_1, A_2, ..., A_n$. Then we form a rule $A : -A_1, A_2, ..., A_n$. Take a union of all such rules gives us the required set. Moreover, those rules are unambiguous. Conversely, once we have that set of rules, we can reconstruct T. The basic idea is to start with an empty tree and for each rule $A : -A_1, A_2, ..., A_n$ add the subtree with root node A and child nodes $A_1, A_2, ..., A_n$ to the partially constructed tree. Repeat the above step until the partial tree does not grow any more. In the end, we will generate T.

The above procedure of reconstructing T can be automated. It is an important feature of our visualization tool: by importing an appropriate set of rules, the tool can render the skeleton of a TDT. By editing in the graphical interface, it is easy to obtain a TDT with rich contents.

There are a few advantages to generate a TDT by a set of user-defined rules:

1) According to the principle of rule induction [15], in order to check the soundness of a derivation tree, it suffices to verify that each rule is sound. In our setting, soundness means that rules and TDTs are reasonable and they conform to engineering practices.
2) A rule may be applied several times in a tree. Therefore, it is more economical to store a set of rules instead of storing the nodes and edges of a tree.
3) We use formulas in first-order logic to formalize properties of trustworthiness. In particular, if the rules can be specified by Horn clauses, we can then resort to Prolog for automated logical reasoning.

Let us take a concrete example to illustrate the construction of a TDT from a set of rules and the logical reasoning based on Prolog.

Example 1: By importing the set of rules given in Figure 2, our tool (to be introduced in Section IV) can render the tree in Figure 3. One could also add some axioms to those rules and make use of Prolog to check if the root property holds or not. For example, Figure 4 shows in a panel the verification result using SWI-Prolog [2]. The target property '*trustworthiness*' does not hold. The reason is that the property '*existence of negative cases list*' cannot be verified, so all the properties along the following path do not hold.

'*existence of negative cases list*' $\longrightarrow$
'*appropriate treatment for known negative cases*' $\longrightarrow$
'*sufficient analysis and design*' $\longrightarrow$
'*trustworthiness in processes*' $\longrightarrow$
'*trustworthiness*'

Notice that a TDT provides a verification framework. The property at each node is a formula as in Example 1. The validity of the root property can be inferred from that of each leaf node. There are two important problems to be considered.

- How to ensure that the rules for constructing the whole tree are sound?

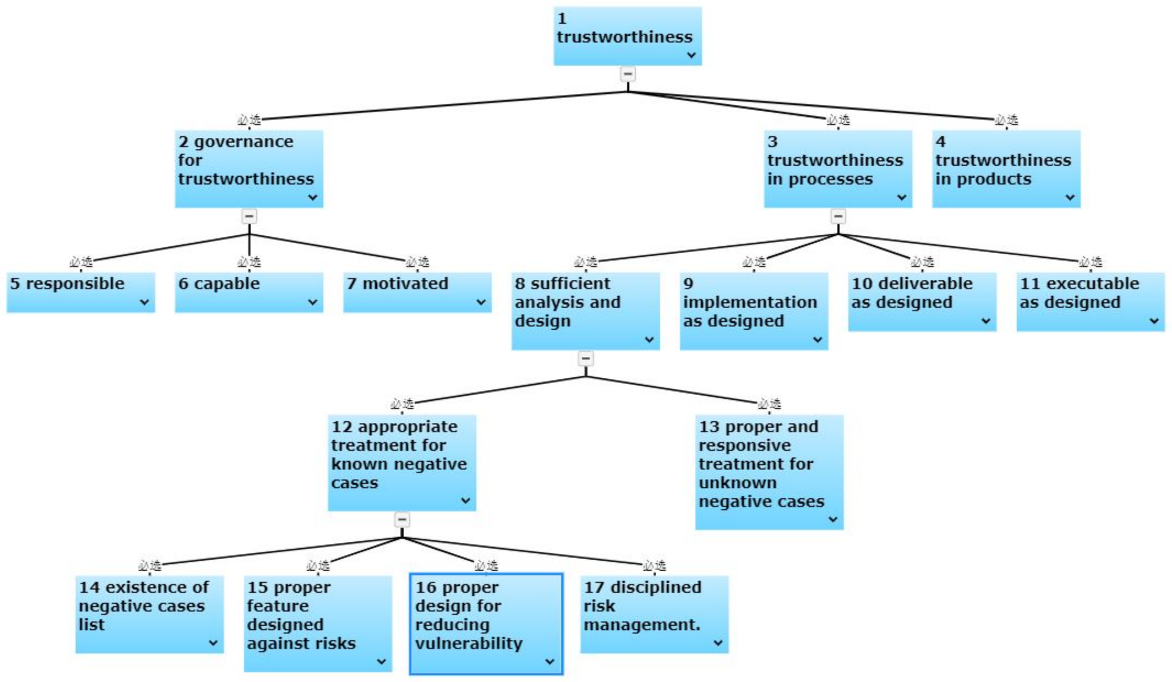

1 trustworthiness
2 governance for trustworthiness
3 trustworthiness in processes
4 trustworthiness in products
5 responsible
6 capable
7 motivated
8 sufficient analysis and design
9 implementation as designed
10 deliverable as designed
11 executable as designed
12 appropriate treatment for known negative cases
13 proper and responsive treatment for unknown negative cases
14 existence of negative cases list
15 proper feature designed against risks
16 proper design for reducing vulnerability
17 disciplined risk management.

Fig. 3: An example TDT

- How to verify the validity of the property at each leaf node?

The model of TDT imposes no restriction for the above two problems. In other words, we can take advantage of other methods to verify the soundness of rules and the validity of the properties at leaf nodes. Sometimes it is hard to argue for the soundness of a rule coming from industrial practice by using formal methods. Then we may resort to Toulmin argumentation model [13] for that purpose. In summary, we adopt first-order logic for reasoning about the property at the root node, but for each rule and the property at each leaf node other formal methods and even informal argumentation are allowed.

IV. Visualization tool

We have developed in Python a visualization tool that provides a graphical user interface to render TDTs in the form of finite trees. The main functionalities of the tool include:

- adding nodes in the tree;
- deleting nodes;
- editing the content of a node;
- importing a Prolog file to render the skeleton of a tree;
- exporting the skeleton of the current tree as a Prolog file;
- saving all the contents of a TDT in a CSV file, which can be reopened later to recover the TDT.

Figure 5(a) provides a bird's eye view of a complete TDT for checking the consistency of software constructions and one subtree is given in Figure 5(b). In our tool the TDT can be zoomed in or out. By clicking on a node, we can hide or display the content in that node, or the subtree rooted at that node. We use solid lines to stand for the full support provided by child nodes to parent nodes, and dashed lines to mean partial support, i.e., the development is not finished yet. When refining the trustworthiness property of a node, we roughly take one of the following two options:

- refining by attributes, which means to refine an attribute mentioned by the property in the node into different components;
- refining by processes, which means to follow a process mentioned by the property in the node and refine it into several steps, as well as its environmental and technical requirements.

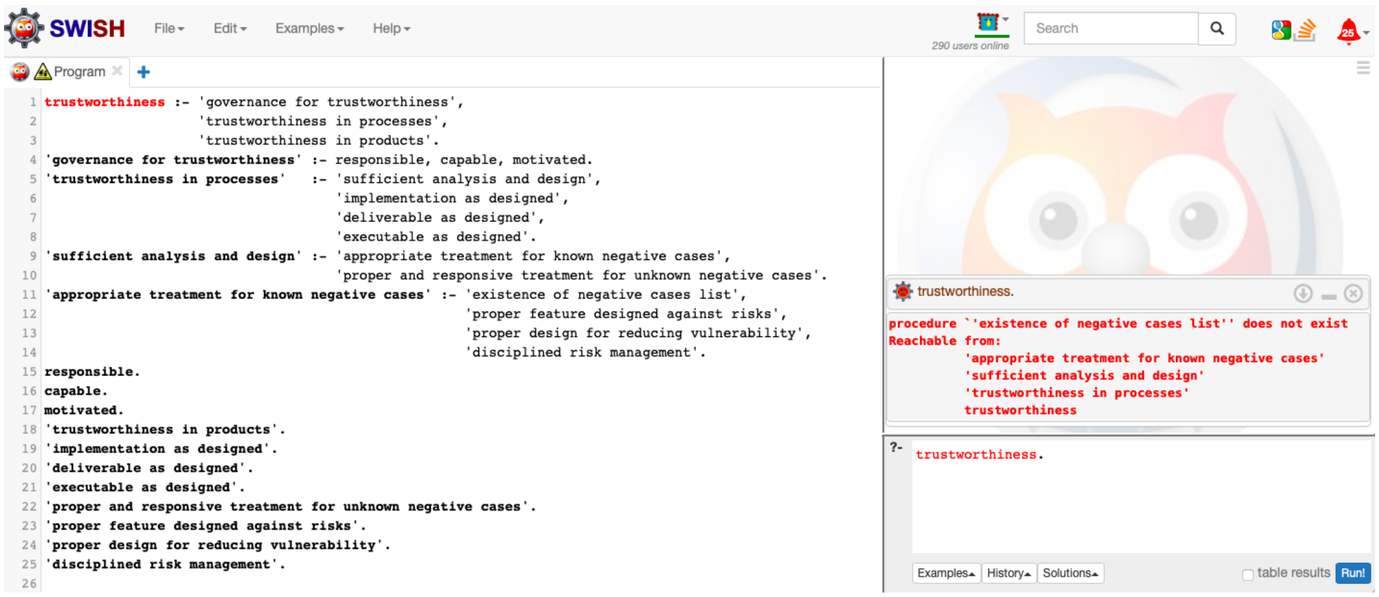
Other types of refinements are possible [5], though they are used less often. In the leaves we store evidences, such as regulatory documents, user manuals, log files etc.

V. Using TDTs in software engineering life cycle

Formal methods have been used in various aspects of software engineering, such as requirement engineering, software design, code verification, testing and so on [16]. They play an important role in improving the quality of products. However, existing formal techniques have limitations in expressing trustworthiness, which is a sociological concept. Our argumentation for trustworthiness is still conducted in a semi-formal way.

In our opinion, TDTs can play a prominent role in the following two phases of a software development life cycle:

- **Phase 1.** Requirement analysis and definition. TDTs serve as a communication means for different stakeholders to agree on the properties about a system. Product definers keep rationally discoursing with customers on how to achieve risk mitigation until a consensus is



```
trustworthiness :- 'governance for trustworthiness',
                   'trustworthiness in processes',
                   'trustworthiness in products'.
'governance for trustworthiness' :- responsible, capable, motivated.
'trustworthiness in processes'   :- 'sufficient analysis and design',
                                    'implementation as designed',
                                    'deliverable as designed',
                                    'executable as designed'.
'sufficient analysis and design' :- 'appropriate treatment for known negative cases',
                                    'proper and responsive treatment for unknown negative cases'.
'appropriate treatment for known negative cases' :- 'existence of negative cases list',
                                                    'proper feature designed against risks',
                                                    'proper design for reducing vulnerability',
                                                    'disciplined risk management'.
responsible.
capable.
motivated.
'trustworthiness in products'.
'implementation as designed'.
'deliverable as designed'.
'executable as designed'.
'proper and responsive treatment for unknown negative cases'.
'proper feature designed against risks'.
'proper design for reducing vulnerability'.
'disciplined risk management'.
```

```
trustworthiness.
procedure `'existence of negative cases list'' does not exist
Reachable from:
          'appropriate treatment for known negative cases'
          'sufficient analysis and design'
          'trustworthiness in processes'
          trustworthiness
```

```
?- trustworthiness.
```

Fig. 4: Verification with SWI-Prolog

reached. This process is supported by Toulmin model of argument.

- **Phase 2.** Product validation. Based on the relevant evidences accumulated in the product development process as well as the product itself, we can use TDTs for deductive reasoning so as to verify whether the product achives trustworthiness. Since a TDT constitutes a structured set of arguments for the property at the root node, and the leaves stand for all the evidences, a TDT is a type of evidence-based model of system trustworthiness. This idea of organizing arguments shares some similarity with Alexy's theory of legal argumentation [3].

In Phase 1, in particular, TDTs can be used for communication between suppliers, customers and regulatory agencies. They can show that a supplier has addressed the trustworthiness concerns from the customers with sufficient and effective protective measures, and the protective measures are implemented correctly according to the design, the related risks have been treated and mitigated, and the relationship between evidences and argumentation can support auditing, communication, argumentation and proofs. More concretely, TDTs are also used to expose the technical goals of a product at the stage of system design, to demonstrate the integrity, consistency, and confidentiality of the process artifacts in managing the development process, to make improvements based on the problems found in construsting the TDTs, and to retain the evidences required for the argumentation. The evidences can consist of presentable facts coming from various abstraction levels or sources, e.g., verification and validation results, configuration files for the development environment and supporting tools, certifications etc.

Up to now, we have piloted the approach of using TDTs in more than a dozen real scenarios such as checking the consistency of software constructions and the trustworthiness of software implementation. Indeed, using TDTs helped us to discover and then resolve some problems. For example, in checking the consistency of software constructions, tag changes in code repositories need to be well managed and file attributes need to be considered in comparing files. One main challenge we have encountered is how to refine a property rigorously. We have empirically identified two types (refinement by attributes or processes). Due to the diversity of scenarios more refinement strategies are yet to be discovered.

VI. Conclusion and Future Work

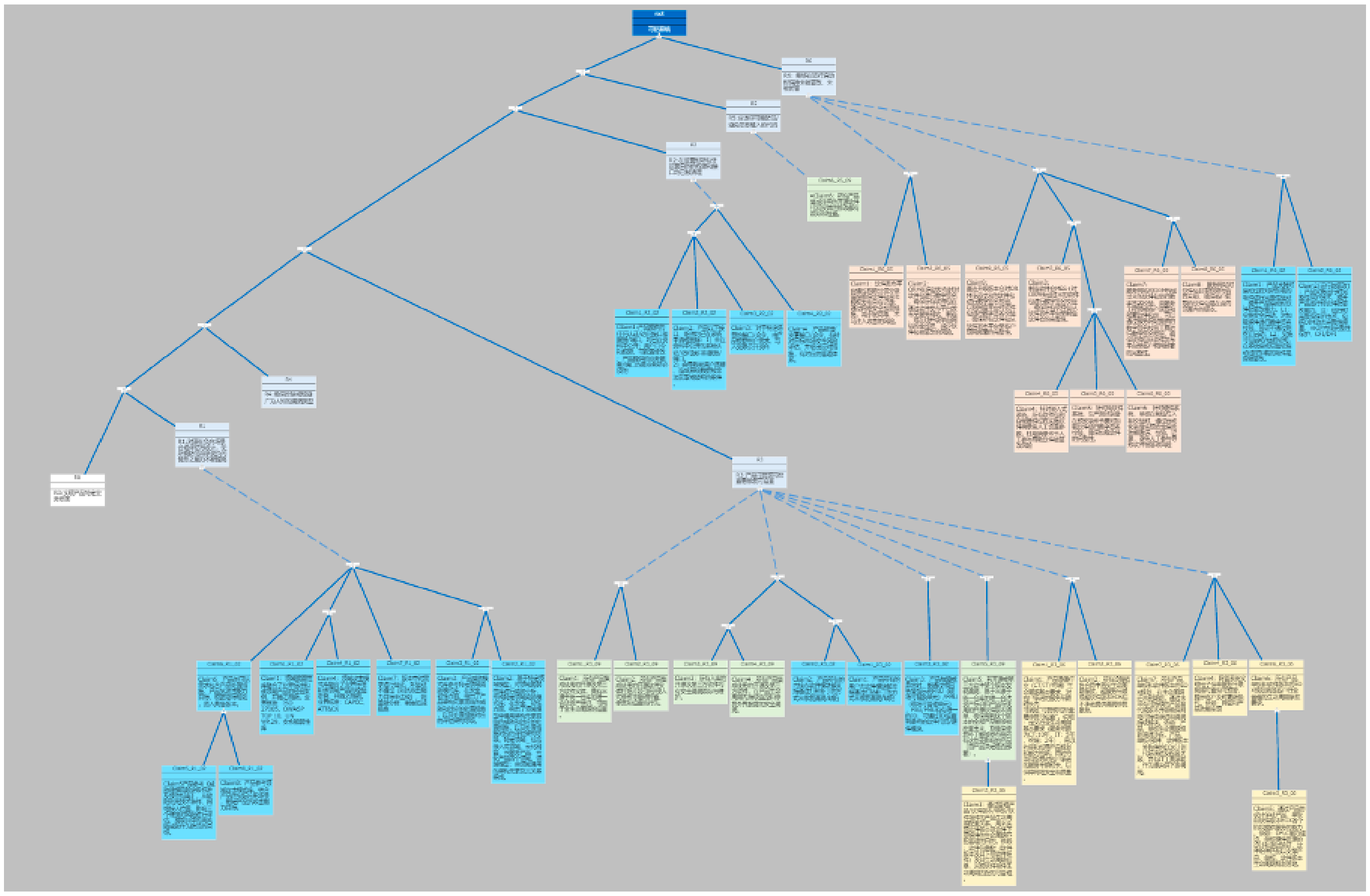
We have presented trustworthiness derivation tree as a model of evidence-based software trustworthiness. The key idea is to gradually refine trustworthiness properties until evidences are reached. Since TDTs can be specified by logical rules, we are able to make use of Prolog to perform automated reasoning. Furthermore, we have developed a visualization tool that allows a user to interactively manipulate TDTs.

We have noticed that evidence-based analysis also appears in legal argumentation [3]. Therefore, we believe our visualization tool could be helpful in the logical reasoning of legal cases.

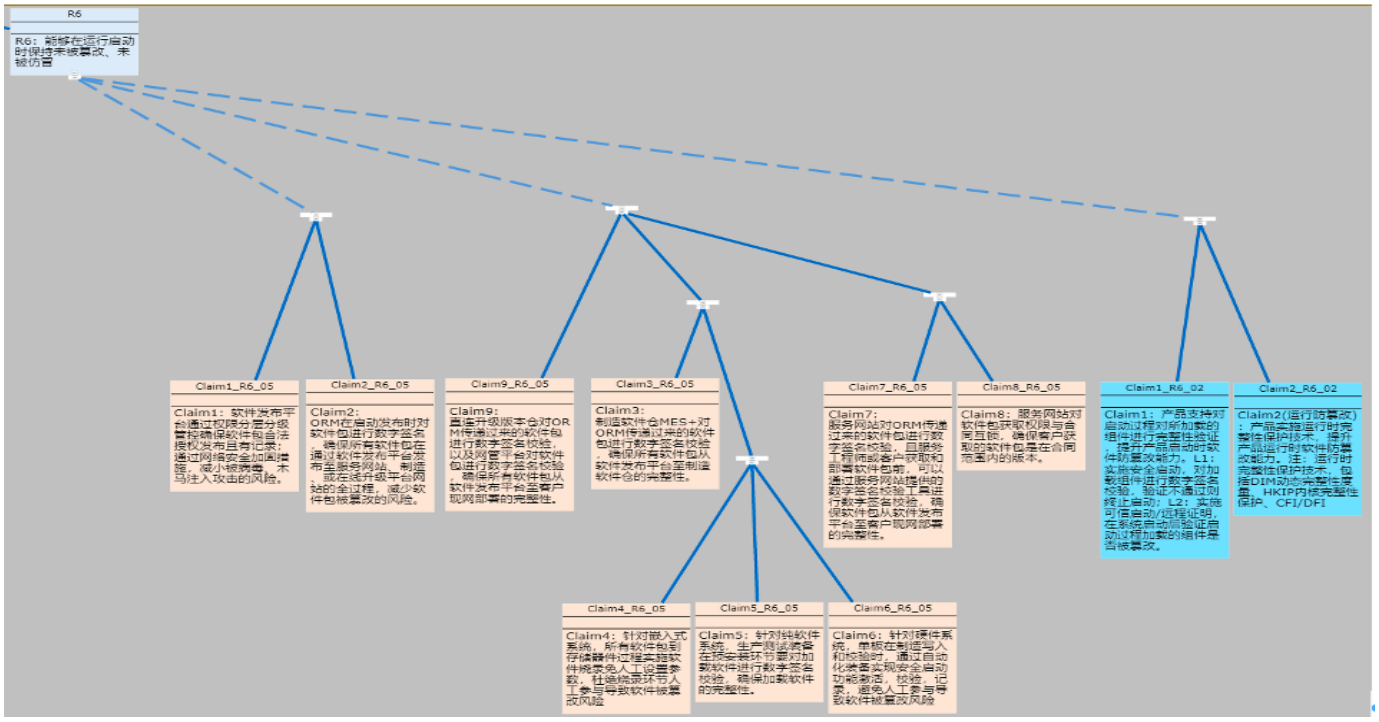
Currently, TDTs are used for qualitative analysis of software trustworthiness. It will be more informative to conduct quantitative analysis if this is possible. In the future, we plan to incorporate quantitative information into TDTs.

Acknowledgment

We thank the anonymous referees for the helpful comments. Deng would like to acknowledge the support of the National Natural Science Foundation of China (Grant Nos. 61832015 and 62072176) and the Inria-CAS joint project Quasar.



(a) Bird's eye view of a complete TDT in the GUI



R6

R6：能够在运行启动时保持未被篡改、未被仿冒

Claim1_R6_05

Claim1：软件发布平台通过权限分层分级管控确保软件包合法授权发布且有记录；通过网络安全加固措施，减少被病毒、木马注入攻击的风险。

Claim2_R6_05

Claim2：ORM在启动发布时对软件包进行数字签名，确保所有软件包在通过软件发布平台发布至服务网站、制造、或在线升级平台网站的全过程，减少软件包被篡改的风险。

Claim9_R6_05

Claim9：直连升级版本会对ORM传递过来的软件包进行数字签名校验，以及网管平台对软件包进行数字签名校验，确保所有软件包从软件发布平台至客户现网部署的完整性。

Claim3_R6_05

Claim3：制造软件仓MES+对ORM传递过来的软件包进行数字签名校验，确保所有软件包从软件发布平台至制造软件仓的完整性。

Claim7_R6_05

Claim7：服务网站对ORM传递过来的软件包进行数字签名校验，且服务工程师或客户获取和部署软件包前，可以通过服务网站提供的数字签名校验工具进行数字签名校验，确保软件包从软件发布平台至客户现网部署的完整性。

Claim8_R6_05

Claim8：服务网站对软件包获取权限与合同互锁，确保客户获取的软件包是在合同范围内的版本。

Claim1_R6_02

Claim1：产品支持对启动过程对所加载的组件进行完整性验证，提升产品启动时软件防篡改能力。L1：实施安全启动，对加载组件进行数字签名校验，验证不通过则终止启动；L2：实施可信启动/远程证明，在系统启动后验证启动过程加载的组件是否被篡改。

Claim2_R6_02

Claim2(运行防篡改)：产品实施运行时完整性保护技术，提升产品运行时软件防篡改能力。注：运行时完整性保护技术，包括DIM动态完整性度量、HKIP内核完整性保护、CFI/DFI

Claim4_R6_05

Claim4：针对嵌入式系统，所有软件包到存储器件过程实施软件烧录免人工设置参数，杜绝烧录环节人工参与导致软件被篡改风险

Claim5_R6_05

Claim5：针对纯软件系统，生产测试装备在预安装环节要对加载软件进行数字签名校验，确保加载软件的完整性。

Claim6_R6_05

Claim6：针对硬件系统，单板在制造写入和校验时，通过自动化装备实现安全启动功能激活，校验，记录，避免人工参与导致软件被篡改风险

(b) A subtree of the TDT in (a)

Fig. 5: A complete TDT

References

[1] http://www.goalstructuringnotation.info/archives/category/resources/tools

[2] https://swish.swi-prolog.org

[3] Alexy, R., Adler, R., MacCormick, N.: A Theory of Legal Argumentation: The Theory of Rational Discourse as Theory of Legal Justification. Oxford University Press (2010)

[4] Bishop, P.G., Bloomfield, R.E.: A methodology for safety case development. In: Industrial perspectives of safety-critical systems. pp. 194–203. Springer (1998)

[5] Bloomfield, R.E., Bishop, P.G.: Safety and assurance cases: Past, present and possible future - an Adelard perspective. In: Proceedings of the 18th Safety-Critical Systems Symposium. pp. 51–67. Springer (2010)

[6] Bloomfield, R.E., Bishop, P.G., Jones, C., Froome, P.: ASCAD – Adelard safety case development manual (1998), Adelard

[7] Bloomfield, R.E., Netkachova, K.: Building blocks for assurance cases. In: Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering Workshops. pp. 186–191. IEEE Computer Society (2014)

[8] International Organization for Standardization: Systems and software engineering – systems and software assurance – Part 2: Assurance case (2011), ISO/IEC 15026-2

[9] Kelly, T., Weaver, R.: The goal structuring notation - a safety argument notation. In: Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases (2004), https://www-users.cs.york.ac.uk/ tpk/dsn2004.pdf

[10] Li, J.H., Mao, B., Liang, Z., Zhang, Z., Lin, Q., Yao, X.: Trust and trustworthiness: What they are and how to achieve them. In: Proceedings of the 5th Workshop on Security, Privacy and Trust in the Internet of Things. IEEE (2021)

[11] Rinehart, D.J., Knight, J.C., Rowanhill, J.: Current practices in constructing and evaluating assurance cases with applications to aviation. Tech. Rep. CR-2015-218678, NASA (2015), https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20150002819.pdf

[12] Rinehart, D.J., Knight, J.C., Rowanhill, J.: Understanding what it means for assurance cases to "work". Tech. Rep. CR-2017-219582, NASA (2017), https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20170003806.pdf

[13] Toulmin, S.E.: The Uses of Argument (updated edition). Cambridge University Press (2003)

[14] William F. Clocksin, C.S.M.: Programming in Prolog. Springer (2003)

[15] Winskel, G.: The Formal Semantics of Programming Languages – An Introduction. MIT Press (1993)

[16] Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: Practice and experience. ACM Computing Surveys **41**(4), 19:1–19:36 (2009)