SafetyADD: A Tool for Safety-Contract Based Design

Fredrik Warg, Benjamin Vedder, Martin Skoglund, and Andreas Söderberg
SP Technical Research Institute of Sweden
Box 857, SE-501 15 Borås, Sweden
Email: {fredrik.warg, benjamin.vedder, martin.skoglund, andreas.soderberg}@sp.se

Published in:

2014 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)

pp. 527 – 529, 2014, IEEE.

Presented at:

4th edition of the IEEE International Workshop on Software Certification (*WoSoCER 2014*), held in conjuction with the 25th IEEE International Symposium on Software Reliability Engineering (*ISSRE 2014*), Naples, Italy, 3-6 Nov. 2014.

DOI: 10.1109/ISSREW.2014.18

SafetyADD: A Tool for Safety-Contract Based Design

Fredrik Warg, Benjamin Vedder, Martin Skoglund, and Andreas Söderberg
SP Technical Research Institute of Sweden
Box 857, SE-501 15 Borås, Sweden
Email: {fredrik.warg, benjamin.vedder, martin.skoglund, andreas.soderberg}@sp.se

***Abstract*—SafetyADD is a tool for working with safety contracts for software components. Safety contracts tie safety related properties, in the form of guarantees and assumptions, to a component. A guarantee is a property the component promises to hold, on the premise that the environment provides its associated assumptions. When multiple software components are integrated in a system, SafetyADD is used to verify that the guarantees and assumptions match when there are safety-related dependencies between the components.**

The initial goal of SafetyADD is to investigate how safety contracts can be managed and used efficiently within the software design process. It is implemented as an Eclipse plugin. The tool has two main functions. It gives designers of software components a way to specify safety contracts, which are stored in an XML format and shall be distributed together with the component. It also gives developers who integrate multiple software components in their systems a tool to verify that the safety contracts are fulfilled. A graphical editor is used to connect guarantees and assumptions for dependent components, and an algorithm traverses all such connections to make sure they match.

I. Introduction

The use of safety contracts has been proposed as a way to build safety cases for systems composed of re-usable software components [1], [7]. When safety standards such as ISO 61508:2010 or ISO 26262:2011 are used, extensive development processes have to be followed, and typically scrutinized by an independent assessor in order to achieve a certification. If software components from a supplier are used, or components from existing products reused, they have to be re-examined as part of the new system, even if they have previously been certified as part of another product.

Safety-contract based design is a proposed methodology that can eliminate the need to re-examine software components for every product where they are used [6], [7]. With this methodology, safety elements out of context (SEooC) components are certified and accompanied with a safety contract that specifies under which circumstances the certification is valid. The safety contract specifies which results the component guarantees under which assumptions. If the component is used within a system that can deliver the specified assumptions, with a certain integrity level, the component is certified to produce the stated guarantees, with a certain integrity level.

SafetyADD is a prototype tool which can be used to create and handle safety contracts, and to verify that systems that integrate multiple components with safety contacts are designed so that the contracts match. When integrated components have safety-related dependencies, the relevant assumptions and guarantees of the components' safety contracts are connected. For instance, if a component has an assumption that an incoming signal must have a value that is not to high, this assumption is connected to a corresponding guarantee on the component that produces the signal. When the system design is ready, an algorithm traverses guarantee-assumption pairs to check that there are no unfulfilled assumptions or mismatches. A mismatch occurs if the specified failure state or integrity level for an assumption is not the same as the guarantee it is connected to. Enabling the use of safety contracts in the software design flow is a step towards safety contract based design using pre-certified software components.

This research is conducted as part of the SafeCer project[1]. Other partners investigate means of defining component contracts using formal methods based on temporal logic, and have developed the tool OCRA [3]. It can check the correctness of contract refinement. Other related tools: AGREE [4], which uses assume-guarantee and compositional verification with temporal logics on architectural models. GAGE [2], a method and tool that maps safety case arguments against a system architecture model, and can check which arguments are satisfied in the model. This tool does not consider SEooC components.

The rest of the paper is organized as follows: Section II discusses the concepts of software component and safety contract. Larger systems are constructed by integrating multiple components; this is described in Section III. Some additional details about the internal workings of SafetyADD can be found in Section IV.

II. Software Components and Safety Contracts

In the context of safety contracts, a software component is a well-defined piece of software designed for reuse. Its functionality, performance, dependencies and possible side-effects are known, so that the safety-related properties can be expressed as a safety contract. The safety contract consist of a number of guarantees and assumptions, and a mapping between guarantees and assumptions. That is, each guarantee contains a list of assumptions that must be valid for the specific guarantee to hold. Fig. 1 shows an example contract with guarantees mapped to assumptions.

Each guarantee and assumption will have a failure state and an associated safety integrity level (SIL). For instance, guarantee 1 in the picture could describe a property of a signal,

[1]See http://www.safecer.eu

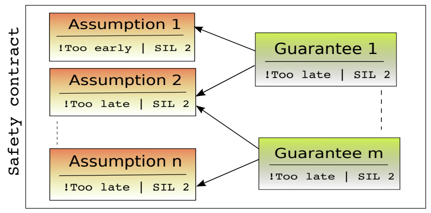

Fig. 1. Connecting guarantees and assumptions.

where the component guarantees that the signal does not enter the failure state `too late` with SIL 2, provided that the required input does not have the failure state `too late` and does not have the failure state `too early`, both also with SIL 2. SafetyADD provides a default list of failure states when defining new assumptions or guarantees; these are: too low, too high, too early, too late, omission, and commission. It is also possible to add new failure states to adapt the tool if needed. It is the subject of our ongoing research to develop an improved scheme to express the failure states.

We call a software component with a safety contract a *primitive component*. After assessment, a primitive component may become a pre-certified SEooC component. Primitive components shall be treated as non-decomposable units in the safety-contract based design process; this is necessary if the validity of certification is to be maintained.

III. Composite Components and Verification

When constructing a software system out of software components, the designer typically has to integrate a number of existing components, and often also develop new ones. SafetyADD handles the safety-contract related activities when designing the system. This is done in the composite component editor. For each software component in the design, a representation of the component's contract is instantiated within a *composite component* in the editor. Within the composite component, assumptions should be connected to matching guarantees for all safety-related dependencies between the components. This is later verified by a contract matching algorithm. A composite component is thus a collection of (primitive of composite) components with connections between the assumptions and guarantees of those components' safety contracts[2].

A. Composite Component Editor

Fig. 2 shows an example composite component in the composite component editor. The large rectangle named *myCompositeComponent* shows the boundary of a composite component. Within the composite component are representations of the containing components' contracts, where round ports represent assumptions and square ports represent guarantees. Within the composite component, assumptions are connected to guarantees on other components, for instance the connection between `comp1` and `comp2` in Fig. 2. Assumptions that are not provided within the composite component can be exported through ports on the composite component border, such as the connection between `comp1` and `myCompositeComponent`. The same is true for guarantee ports. The connectors between ports on internal components and the composite component border are called *delegation connectors* (we use the same terminology as the unified modeling language).

The composite component editor makes it easy to rearrange a design or replace a component. It it also possible to maintain several variants of the design, saved as separate composite components. In addition, it is easy to visually spot unconnected assumptions and get an overview of the safety-related dependencies between components. SafetyADD currently only supports primitive components within a composite component. Future work is to also support nested composite components for increased composability.

B. Safety-Contract Verification

There are two ways safety contracts are checked. The most basic check matches the failure state and integrity level for each connection within the composite component, and lists unconnected assumptions and connections where failure state or SIL do not match.

It is also possible to create test-cases for specific use-cases. For instance, consider a situation where a composite component has multiple guarantees and assumptions, but the designer wants to use it in an environment that only fulfills some of the assumptions, and only uses one of the guarantees. In this case, it is necessary to check if the wanted guarantee holds given the available assumptions. Such test cases uses specific *test components* that can be connected to the ports on the composite component. Fig. 2 has two such components: `test1` and `test2`. Test components are essentially dummies with the single purpose of indicating for the test engine which assumptions and guarantees to include for this specific use-case. When running the test, SafetyADD will treat the ports on the composite component under test as follows:

- A guarantee connected to a test component means that the test-case will check if this guarantee holds.
- An unconnected guarantee will not be included in the test.
- An assumption connected to a test component means that the environment[3] is expected to provide this property.
- An unconnected assumption is treated as an assumption the environment will *not* fulfill.

When validating a design with test components, the verification algorithm will start at each guarantee connected to a test component, and work its way through the connections. In Fig. 2, `test1` is connected to the guarantee port on `comp2` via a delegation connector. The algorithm will check the safety contract of `comp2` to determine which assumptions are needed to fulfill this particular guarantee, and then proceed to check whether those assumptions are connected to a valid guarantee. This will continue recursively until all paths have terminated. Unless all paths terminate either internally in a primitive component, or externally in an assumption connected to a test component, there will be a verification error. Part of the algorithm is expressed in pseudo-code below.

[2]SafetyADD largely follows the SafeCer component model which also uses the terms primitive and composite component [5].

[3]The environment, in this case, would be other software or hardware the component is designed to interact with.

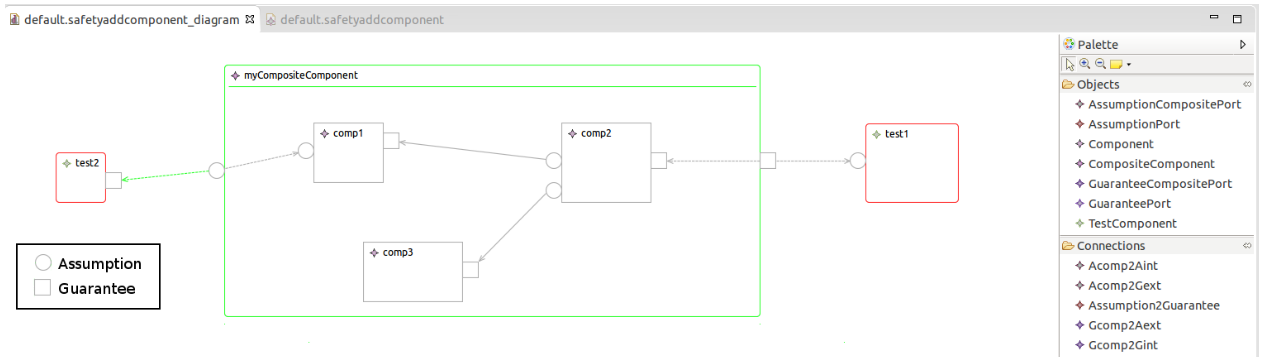

Fig. 2. A composite component with test stubs.

Pseudocode – path traversal algorithm

```
G = set of <test component guarantees>;
foreach g in G {
  test_guarantee(g);
}

test_guarantee(g) {
  A = set of <assumptions required by> g;
  foreach a in A {
    g_next = <guarantee connected to> a;
    assert(g_next != NULL);
    if(!is_test_component(g_next)) {
      assert(contract_match(a,g_next));
      test_guarantee(g_next);
    }
  }
}
```

IV. SafetyADD: Under the Hood

SafetyADD is developed as a plugin to the Eclipse[4] integrated development environment (IDE). This has several important benefits. It makes it possible to take advantage of existing Eclipse frameworks such as Eclipse Modeling Framework (EMF), Graphical Modeling Framework (GMF) and Epsilon to create a graphical interface with a look-and-feel software developers are familiar with. It also makes it possible to create a single integrated environment for developing the software components themselves, and working with safety contracts. System requirements are Eclipse Luna (4.4) and Java SE 7 or later on any Eclipse supported platform.

The creation of safety contracts are made using a form where details about the component can be specified; most importantly the assumptions and guarantees. The safety contracts are stored in an XML format with object-oriented inspired design. The format is under development and will evolve. Currently there are classes for primitive components, composite components, assumptions, guarantees, and test components. Composite components contain instances of primitive components, delegation and regular connections. Primitive components contain contracts, which in turn contain instances of guarantees and assumptions (these are shown as ports on the primitive component in the component editor). Assumption and guarantee classes contain the properties used in contract verification. The safety contracts are not directly tied to the component code, and are therefore language agnostic when it comes to the component implementation.

[4]See http://eclipse.org

V. Conclusion and Future Work

This paper presented SafetyADD, a tool used to create and handle safety contracts for software components, and validate designs of multiple integrated software components with such safety contracts. We are currently working on refining the contract specification and matching mechanisms. Possible improvements are to support nested composite components, SIL decomposition, OR-ing of assumptions, an improved way to express failure states, and mechanisms to handle configurable components. Another future improvement is to add the possibility to digitally sign contracts and tie them to a verified implementation; this functionality can be part of making precertification of software components with safety contracts a reality.

Acknowledgment

The authors would like to thank Danesh Daroui for his contribution to SafetyADD. This work has been funded by the research project nSafeCer within the ARTEMIS JU programme under grant agreement no 269265, and VINNOVA - Swedish Agency for Innovation Systems.

References

[1] I. Bate, R. Hawkins, and J. McDermid. A contract-based approach to designing safe systems. In *8th Australian Workshop on Safety Critical Systems and Software (SCS 2003)*, volume 33 of *CRPIT*, pages 25–36. ACS, 2004.

[2] Stefan Björnander, Rikard Land, Patrick Graydon, Kristina Lundqvist, and Philippa Conmy. A method to formally evaluate safety case arguments against a system architecture model. In *2nd edition of the IEEE Workshop on Software Certification (WoSoCER2012)*. IEEE Computer Society, November 2012.

[3] A. Cimatti, M. Dorigatti, and S. Tonetta. OCRA: A tool for checking the refinement of temporal contracts. In *ASE*, pages 702–705, 2013.

[4] Darren D. Cofer, Andrew Gacek, Steven P. Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, volume 7226, pages 126–140. Springer-Verlag, April 2012.

[5] J. Carlson et al. Generic component meta-model, 2012. http://www.safecer.eu/text/view/50/.

[6] A. Söderberg and R. Johansson. Safety contract based design of software components. In *International Symposium on Software Reliability Engineering (ISSRE 2013) (Supplemental Proceedings)*, pages 365–370, 2013.

[7] A. Söderberg and B. Vedder. Composable safety-critical systems based on pre-certified software components. In *International Symposium on Software Reliability Engineering (ISSRE 2012) Workshops*, pages 343–348, 2012.