Deriving Safety Cases for Hierarchical Structure in Model-Based Development

Nurlida Basir[1], Ewen Denney[2], and Bernd Fischer[1]

[1] ECS, University of Southampton, Southampton, SO17 1BJ, UK
(nb206r,b.fischer)@ecs.soton.ac.uk
[2] SGT / NASA Ames Research Center
Moffett Field, CA 94035, USA
Ewen.W.Denney@nasa.gov

Abstract. Model-based development and automated code generation are increasingly used for actual production code, in particular in mathematical and engineering domains. However, since code generators are typically not qualified, there is no guarantee that their output satisfies the system requirements, or is even safe. Here we present an approach to systematically derive safety cases that argue along the hierarchical structure in model-based development. The safety cases are constructed mechanically using a formal analysis, based on automated theorem proving, of the automatically generated code. The analysis recovers the model structure and component hierarchy from the code, providing independent assurance of both code and model. It identifies how the given system safety requirements are broken down into component requirements, and where they are ultimately established, thus establishing a hierarchy of requirements that is aligned with the hierarchical model structure. The derived safety cases reflect the results of the analysis, and provide a high-level argument that traces the requirements on the model via the inferred model structure to the code. We illustrate our approach on flight code generated from hierarchical Simulink models by Real-Time Workshop.

Keywords: Model-based software development, automated code generation, formal proofs, formal analysis, safety case, automated theorem proving.

1 Introduction

Model-based development and automated code generation are increasingly used for actual production code, in particular in mathematical and engineering domains. For example, NASA's Project Constellation uses Real-Time Workshop (RTW) for its Guidance, Navigation, and Control (GN&C) systems and subsystems. However, since code generators are typically not qualified, there is no guarantee that their output is correct or even safe, and additional evidence of its safety is required. In previous work [5], we have thus constructed safety cases [19] from information collected during a formal verification of the generated code. We also have constructed safety cases that correspond to the formal proofs found by automated theorem provers of the verification conditions, and reveal the underlying proof argumentation structure and top-level assumptions [6].

This paper is a continuation of our previous work, but here we systematically derive safety cases that argue along the hierarchical structure in model-based development.

E. Schoitsch (Ed.): SAFECOMP 2010, LNCS 6351, pp. 68–81, 2010.

A safety case is a structured argument, supported by a body of evidence, which provides a convincing and valid justification that a system is acceptably safe for a given application in a given operating environment [19]. In the Goal Structuring Notation (GSN) [13], which we use as technique to explicitly represent the logical flow of a safety argument, the main construction elements of a safety case are goals (which are the safety claims to be met by the system), strategies (which describe how a claim is addressed by evidence or further subgoals), evidence, and assumptions. In our work, the safety cases are constructed mechanically using a formal analysis, based on automated theorem proving, of the automatically generated code. Goals are thus given by the formal safety requirements on the model, which express as logical formulas the properties that the (software sub-) system's output signals must satisfy for the (overall) system to be safe. Strategies are the high-level steps of the formal analysis (e.g., decomposing the set of requirements, or decomposing the system into components) while the evidence comes from the low-level proofs of the verification conditions. Assumptions are logical formulas which express the properties that the input signals must satisfy for the (overall) argument to be valid; they are thus dual to requirements and, hence, goals.

We illustrate our work using the verification of two safety requirements for a spacecraft navigation system that was generated from a Simulink model by Real-Time Workshop [3]. The requirements determine the interface between the software system safety cases (where they are the root elements) and the subsystem safety case (where they are leaf nodes). Each requirement induces a verified abstraction or *slice* of the system architecture. The formal analysis recovers the hierarchical structure of these slices from the code and identifies requirements that rely on any externally given assumptions. This enables us to identify how the system safety requirements are broken down into low-level component requirements and distributed over the system components, and thus also to identify where the requirements are ultimately established, resulting in a hierarchy of requirements that is aligned with the hierarchy of the components.

We use safety cases to reflect the results of the program analysis, and provide a high-level argument that explains how the system slices establish the corresponding safety requirements. The safety cases help tracing the safety requirements from the model via the inferred system structure to the code, thus providing independent assurance of both model and code. They also provide a traceable safety argument that shows in particular where the code, subsystem, and system depend on any internal and external assumptions. We believe they highlight the claims, key safety requirements, and evidence that are required to understand and trust generated code, which is essential for the use of code generators in safety-critical applications.

2 Background

2.1 Model-Based Software Development

Model-based software development comprises a number of techniques that focus on creating and transforming domain-specific abstractions or *models* rather than algorithms or even code. In *model-based design* [3,18], mathematical or, more often, visual methods are used to create an initial model of the system design. It is commonly

used in the control systems domain, where block diagrams provide an accepted notation. Blocks can represent arbitrary computations and can be nested hierarchically, which helps counter system complexity. They are connected by wires that represent the flow of signals through the system. A number of academic and commercial tools support model-based design in this domain. We focus on MathWorks Simulink [3], which is used by many NASA projects for at least some of their modeling and code development, particularly for GN&C problems. Simulink comes with a large library of standard modeling blocks that provide mathematical operations and signal routing suitable for control systems and complex operations.

Model-based code generation [16,18] complements model-based design, and translates specifications in the form of a model into a program in a high-level programming language such as C or ADA. The translation process can be organized as a sequence of model transformations, where the last model is equivalent to the program. The final source code generation can then be realized with a simple template engine. Here we focus on a commercial generator, MathWorks Real-Time Workshop Embedded Coder [3]. Real-Time Workshop generates ANSI/ISO compliant C and C++ code from MathWorks Simulink and Stateflow models. Embedded Coder adds various features, such as optimization, which are useful for generating C code tuned for embedded devices.

2.2 Formal Program Analysis Using AutoCert

The techniques described here are based on the AUTOCERT code analysis tool [9], which takes a set of requirements, and formally verifies that the code satisfies them. AUTOCERT can verify execution-safety requirements (e.g., array bounds), as well as individual mathematically specified requirements. AUTOCERT thus supports certification by formally verifying that auto-generated code is free of certain safety violations and complies with domain-specific safety requirements as those mentioned.

AUTOCERT follows the Hoare logic approach to verification, which needs *annotations*, i.e., logical assertions of program properties, at key locations in the code. These annotations are constructed automatically by a post-generation inference phase that exploits the idiomatic nature of auto-generated code and is driven by a generator- and domain-specific set of idioms. The inference algorithm builds an abstracted control-flow graph (CFG), collapsing the code idioms into single nodes. It then traverses the CFG from *use* nodes (where a requirement must hold) backwards to all corresponding *definitions* (where the relevant properties are established) and annotates the statements along the paths as required [9]. The definitions typically correspond to model blocks (more precisely, to parts of the code implementing a block), which can use assumptions on the properties of their input signal to establish the requirement. Hence, the inference algorithm must recurse over the variables corresponding to the input signals, derive the assumptions, and establish them as new requirements. This chain of requirements on variables and their definitions constitutes the backbone of our safety argument. As byproduct, the inference derives the component interfaces (i.e., the requirements placed on them, and the assumptions made by them) as well as the system's overall assumptions, which need to be established by its context. A verification condition generator (VCG) processes the annotated code, feeding a set of verification conditions (VCs) into an automated theorem prover (ATP); their proofs guarantee that the code satisfies the

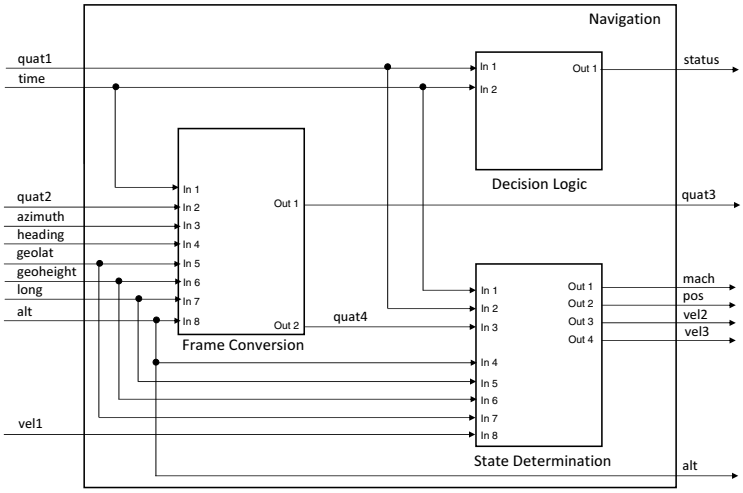

Fig. 1. High-level Architecture of Navigation System

requirements and also validate the definitions identified by the analysis, and thus the derived architecture. In the safety case, the proofs serve as evidence.

During the course of analysis, AUTOCERT records various facts, such as the locations of uses and definitions, which are later used as input to the safety case generation process. Here, we extended the existing mechanism to record additional information from which we can reconstruct the system architecture slices.

2.3 Guidance, Navigation, and Control Systems

Spacecraft are typically decomposed into a number of different systems such as the power, thermal protection, or guidance, navigation, and control (GN&C) systems [22]. The GN&C system is a necessary element of every spacecraft. Here, we focus on the Navigation (sub-) system within the GN&C system. It is used to determine a spacecraft's orientation and position, which is challenging from a safety perspective, due to its complex and mathematical nature. We give a brief, simplified description of the system where we also have changed the names of components and signals from the original.

Navigation (see Fig. 1 for its architecture) takes several input signals, representing various physical quantities, and computes output signals representing other quantities, such as Mach number, angular velocity, position in a specified frame of reference, and so on. Signals are generally represented as floating point numbers or as quaternions and have an associated physical unit and/or frame which correctness are critical to the safety of the system. However, the units and frames are usually not made explicit in the model, and instead are expressed informally in comments and identifiers.

Navigation is comprised of three components, a decision logic that computes a status value irrelevant to the requirements we consider here, a frame conversion, and a state determination. Frame Conversion first converts the frames of the incoming signals from a vehicle-based coordinate system to an earth-based coordinate system. The transformations of the coordinate systems are done by converting quaternions to direction cosine matrices (DCMs), applying some matrix algebra, and then converting them back to quaternions [20]. State Determination then performs the calculations to determine the vehicle state (e.g., position, attitude, and attitude rate) from these signals. It is defined in terms of the relevant physical equations. Note that there are no individual blocks within Navigation, but only within the components and thus all computation happens there.

3 Deriving Safety Cases from the Formal Analysis of Hierarchical Structure

While Leveson et al. [14] rightly argue that a formal verification of software against its requirements does not guarantee safety, it is important to note that the safety requirements are not the same as the software requirements specification, even if many requirements from a software requirements specification do impact safety. Instead, in this work, we assume that the safety requirements have been established independently, for example by a hazard analysis of the overall system, and so take them as given.

Here we use requirements on the GN&C as driving example, since the GN&C is clearly safety-critical, and maintenance of the correct navigation state is therefore safety-critical. In particular, we require that the navigation state be represented in the correct coordinate frames, where "correct" has been independently determined.

3.1 Formalization of the Requirements

We illustrate our work using the results of the formal verification of two safety requirements for the code generated from the Simulink model of the above navigation system:

(1) The system shall compute a quaternion representing a transformation from the Earth-Centered Inertial (ECI) frame to the body fixed frame in signal $quat_3$, and

(2) The system shall compute a velocity in the ECI frame in signal vel_2.

Since we are working with a formal, logic-based analysis framework, we need to formalize these requirements using a domain theory:

(1′) $quat_3 :: \mathrm{quat}(\mathrm{ECI}, \mathrm{Body})$.

(2′) $vel_2 :: \mathrm{vel}(\mathrm{ECI})$.

Here, ECI and Body are constants denoting the respective frames, quat and vel are functions denoting transformations of or quantities in those frames, and :: is a predicate that asserts that the signal represents a transformation between (resp. quantity in) the required frame(s).

Obviously, the actual formalization of the safety requirements themselves is safety-relevant: a wrong formalization can invalidate the assurance provided by the proofs [4,15]. It thus needs to be called out and justified in the safety case.

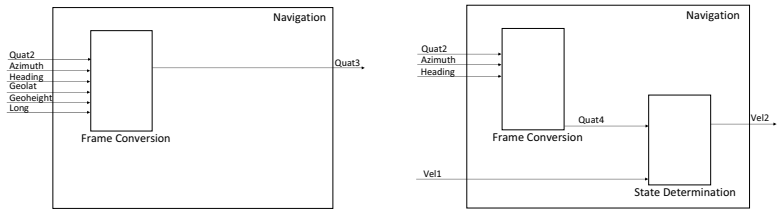

Fig. 2. Architecture Slices Recovered for Example Requirements

3.2 Architecture Recovery

In order to certify the requirements on a system, and to build a comprehensible safety case, we need to know where in the system they are established, and which parts of the system contribute to them. In the system architecture (Fig. 1) we can see that the first requirement should be established by Frame Conversion, since the signal $quat_3$ comes straight out of that component (and similarly for vel_2 and State Determination in the case of the second requirement). However, this view is too simplistic. First, without looking inside the component models it is not clear whether the requirement is indeed established within a component, or simply passed through (cf. for example *alt* in Navigation), and which of the component's input signals (if any), or more precisely which assumptions on them, are used in establishing the requirement. However, simply expanding the component models destroys the hierarchical structure of the system. More importantly, the safety of the system ultimately depends on the safety of the code rather than the model, but because we cannot trust the code generator to translate the model correctly we cannot derive any trust from the model.

Instead, we analyze the code and recover the slice of the system architecture that is relevant to a given safety requirement. We record when the analysis enters resp. leaves a component (implemented by RTW as a parameter-free procedure), and then remove the part of the requirements-definition chain that is contained within the component. The key to obtaining precise architecture slices is to identify situations in which the control flow just passes through a component, without encountering a definition. In these cases, we can ignore the component altogether. We then assemble the slices from the signals involved in the recorded requirements-definitions chains and from the retained component.

Fig. 2 shows the architecture slices recovered for both requirements. In both cases, the irrelevant Decision Logic component has been removed by the analysis. For the first requirement, it has further identified that $Quat_3$ is unaffected by the call to the State Determination procedure, and consequently removed that component as well. For the second requirement, the analysis has identified $Quat_4$ as the (global) variable through which the two components communicate. In addition, although not shown in Fig. 2, it has derived the property placed as an assumption on this variable by State Determination, i.e., $Quat_4 :: \mathrm{quat}(\mathrm{NED}, \mathrm{Body})$. This becomes a subordinate requirement to the

original safety requirement, reflecting the hierarchical model structure. The requirements hierarchy is completed by the assumptions placed on the variables Vel_1 and $Quat_2$ corresponding to the components' input signals.

The property derived for $Quat_4$ also becomes part of the interfaces of both components that are connected through this link, as assumption on the State Determination and as safety requirement on Frame Conversion. By regrouping the analysis results by component rather than by original safety requirement, we thus obtain full component interfaces. They give a complete functional specification of the component, including all assumptions, as far as it is required to satisfy the given system-level safety requirements. The interfaces also serve as starting point for verifying the components independently, hence allowing a compositional (and therefore scalable) verification.

The recovered system architecture and requirements hierarchy already constitute a core safety argument: Navigation satisfies the safety requirement ($2'$) if the components Frame Conversion and State Determination satisfy their respective interfaces, and the requirements for Vel_1, $Quat_2$, and $Quat_4$ hold. This argument can serve as blueprint for a full-fledged safety case. In addition, the derived component interfaces serve as starting points for the construction of independent safety cases for the components, yielding a hierarchy of safety cases that is aligned with the system's hierarchy of models.

3.3 Arguing from System-Level Safety Requirements to Component-Level Safety Requirements

The upper part of the safety case argues the safety of the method of formal reasoning that we use but also points out the important provisos that we abstract away from real-time, and numerical issues. This is a straightforward modification of our previous work on programs without hierarchical system structure (see Fig. 3 Tier I: Explaining the Safety Notion in [5]). Here, we thus focus on the lower part of the safety case that explains that, and how, the generated source code Nav.cpp satisfies the given safety requirements by providing formal proofs as evidence (see Fig. 3).

The key argument strategy here is to argue over each individual requirement that contributes to the program safety. The additional information that is required for the strategy to be understood and valid is identified and explained. This concerns the independent validity of the safety requirements and the logical consistency of the assumptions. We thus assume that no safety requirement is available for use as a (logical) assumption in the safety proofs, which prevents vacuous proofs based on mutually recursive dependencies between requirements and assumptions. We further assume that the given and derived assumptions together are consistent, again to prevent vacuous proofs. Each assumption is justified by a valid justification (e.g., the consistency can be checked by theorem prover).

As a result of this strategy we get as many subgoals as there are safety requirements given. Here we focus on the goal (R2) corresponding to the second requirement, i.e., that the system shall compute a velocity in the ECI frame in signal vel_2. Context nodes with hyperlinks outline additional evidence in the form of documents, containing, for example, a detailed description of the system and requirement, and also the result of the hazard analysis.

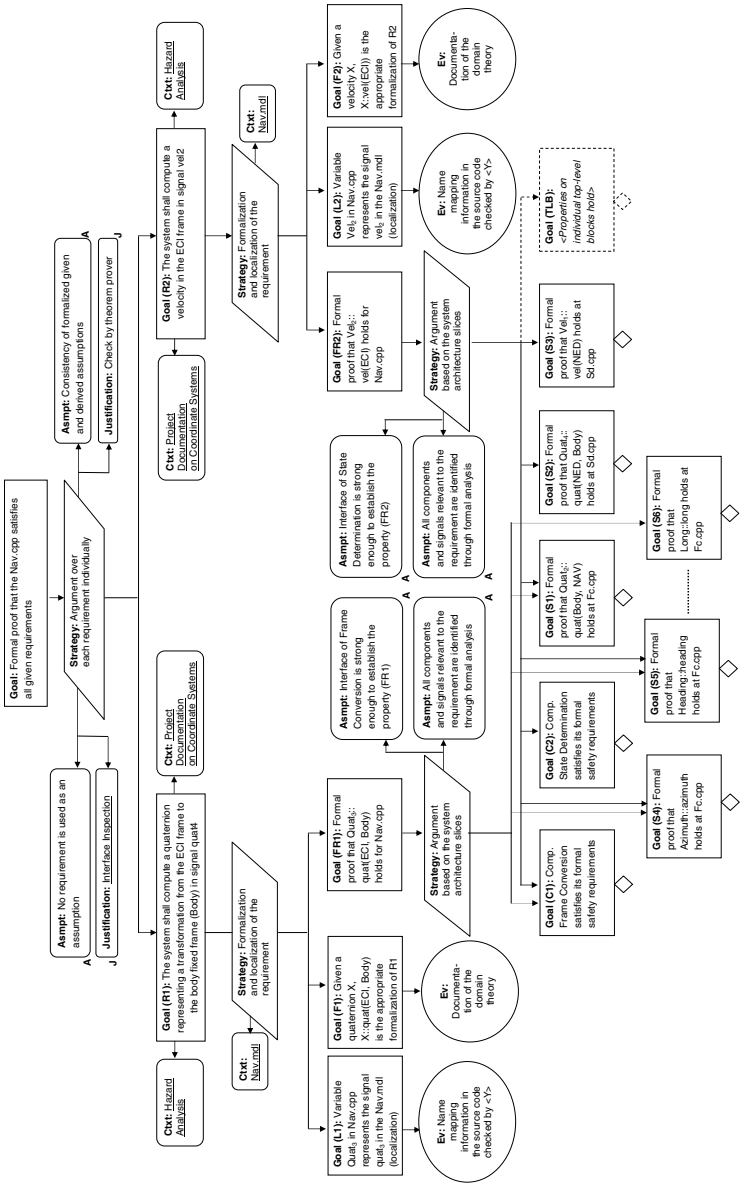

Fig. 3. Arguing from System-Level Requirements to Component-Level Requirements

The next step of the argument transitions from the informal level to a formalized safety requirement. This step helps in showing that the formal verification runs over the correct requirement, based on the right formula and variable, and thus provides a relevant proof of the program. We use an explicit strategy to describe this transition, which spawns three subgoals. As already discussed in Section 3.1, the first subgoal (F2)

demonstrates that the formal proof is based on an appropriate formalization of the requirement, and the safety case points to the documentation of the logical domain theory as evidence of this. The second subgoal (L2) "glues together" model and code levels, which allows us to build a safety case for the model based on the analysis of the code. In particular, as discussed in Section 3.2, we need to show the mapping between the signal names used in the model and the corresponding variable names used in the source code, which cannot be recovered by our analysis but must be given externally. Here, the safety case points to the mapping information given in the source code, and that it has been checked by a reviewer, as evidence. In addition, at this goal we also have to show the mapping between the model and code files, and in particular, in which code file the property formalized in (F2) has to be shown. In our example, this is straightforward, but for larger systems the localization needs more evidence.

With the results of (F2) and (L2) we can now construct the final subgoal (FR2) of our strategy, which shows that the fully formalized safety requirement Vel_2:: vel (ECI) holds after execution of the code in Nav.cpp. This requirement eventually needs to be proven formally. However, at this level of abstraction, the safety case does not use an argument based on the full formal proofs. Instead, we use an argument based on the system architecture, or more precisely, on the recovered system architecture slices. It shows how the system level requirements are broken down into the component level requirements i.e., properties of the part of the system that is relevant to satisfy the requirement (FR2). The strategy is based on the assumption that the formal analysis has identified all relevant components and signals. We thus reduce (FR2) to a number of (delayed) subgoals for the components and signals in the architecture slice. For each component, we need to show that it satisfies the safety requirements specified in its interface (i.e., subgoals (C1) and (C2)). This induces a further assumption on the strategy, namely that the interface is strong enough to show the requirement (FR2). Delaying the subgoals allows us to reuse the component-level safety cases. This way, we achieve a hierarchical structure for the system safety case that mirrors the hierarchy embedded in the system architecture. If the system contains top-level blocks in addition to the components (which is not the case in our example), we need to reason about their properties as well. This is indicated by the dashed subgoal (TLB). For each variable representing a signal, we need to show that it satisfies the safety requirements derived by the analysis (i.e., subgoals (S1) to (S5)). This guarantees that the components' assumptions are met. These subgoals are delayed here as well, to keep the safety case compact. Their expanded structure again follows the lines of our previous work [5], and uses the argumentation shown in Fig. 5 (Tier III) of the safety case there with small modifications; in particular, the notion of safety condition needs to be replaced by that of safety requirement. Note that we make no distinction at this level between subgoals that are established by the components (S2) and those that are reduced to assumptions about the system's input signals and thus have trivial formal proofs, e.g., (S4).

3.4 Arguing from Component-Level Safety Requirements to Source Code

In the next step of our hierarchical development, we argue about the safety of the components wrt. their identified interfaces. The component-level safety cases also argue about a set of requirements, but there are two significant differences to the system-level

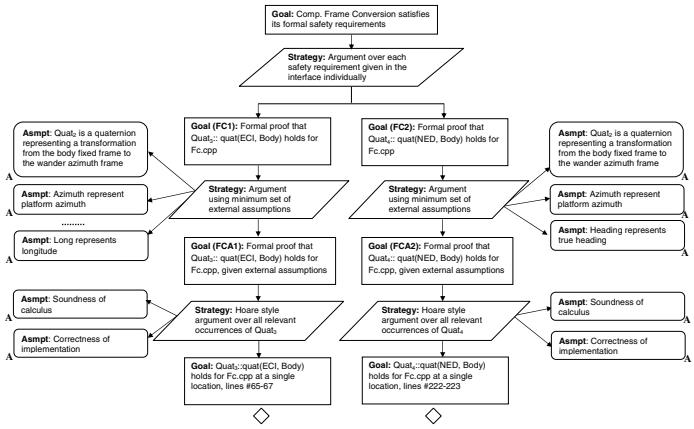

Fig. 4. Component-level safety case for Frame Conversion

safety cases. First, the component-level requirements are already formalized, due to the use of the formal analysis, so that we do not need to argue about the safety of the formalization and localization any more. Second, the argument will generally go down to the level of the generated code, with the proofs of the VCs as evidence; obviously, however, another layer of hierarchy is introduced if a component contains further components.

Fig 4 shows the safety case for the Frame Conversion component. For each component, the strategy is to argue over each individual safety requirement stated in its interface. Here, we have two requirements, (FC1) which is used to discharge the (essentially identical) system-level goal (FR1) via (C1), and (FC2), which is used to discharge the signal subgoal (S2). Even though they serve different purposes in the system-level safety case we treat them the same at the component level. We focus on (FC2) here.

The component interfaces also list the assumptions that the component itself makes about the environment. However, not all assumptions are used for all requirements, so we use an explicit strategy to argue only using the minimal set of external (i.e., on the system's input signals) assumptions. Note that the use of internal assumptions (e.g., on $Quat_4$), which have been identified as subgoals in the system-level safety case (i.e., (S2) in Fig. 3) will be made explicit further down in the component-level safety case.

The next strategy finally transitions from the safety argument to a program correctness proof, using a Hoare-style argument over all relevant occurrences of the variable. The structure of this Hoare-style argument is determined by the structure of the program. In this case, it leads to a single subgoal, proving that the safety requirement holds at the given source location. This is predicated on the assumptions that the applied Hoare-calculus is sound, and that the VCG is implemented correctly, which need to be justified elsewhere. Since the rest of the safety case is constructed as described in our previous work [5], we do not expand it here any further.

Showing the safety of the component is thus reduced to formally showing the validity of the VCs associated with each requirement in the interface. If (and only if) proofs for all corresponding VCs can be found, then the property holds for the entire program. The construction of safety cases from the proofs is described in our previous work [6].

3.5 Combining System-Level and Component-Level Safety Cases

Splitting the argument into system-level and component-level makes it easier to follow and allows common sub-arguments to be factored out, but in order to obtain a complete argument we need to combine the system- and component-level safety cases. However, simply attaching the entire component-level safety cases to the corresponding component goals would introduce redundancies. Clearly, not every safety requirement on the system level relies on the full set of requirements established by the components, for example, (FR2) only uses the requirement derived for $Quat_4$ (i.e., goal (FC2) in Fig 4).

We thus replace each component goal only by the "branches" of the component-level safety case that are required; this information is provided by the program analysis. For component goals that are shared between different requirements this will lead to an "unsharing". For example, (C1) will be replaced by the branch rooted in (FC1) below (FR1) and by the one rooted in (FC2) below (FR2). However, common subgoals at the level of the Hoare-style argument, which are based on computations contributing to different requirements, can remain shared.

Additional changes occur elsewhere in the system-level safety case. The assumptions to the architecture-based strategy solving (FR1) and (FR2) can be removed because the detailed argumentation in the component-level safety case provides the necessary evidence. Further the subgoals associated with the system's input signals (i.e., (S1) and (S3)–(S6)) can be removed because corresponding subgoals still appear as leafs in the component-level safety case, where they are discharged by the assumptions. The subgoals on the connecting signals (here only (S2)) will be replaced by the root goals of the corresponding branches in the component-level safety case (i.e., (FC2)) at the appropriate position in the Hoare-style argument for the client component (i.e., State Determination).

4 Safety Case Construction

The safety cases described here quickly become too large for manual development. Fortunately, the bulk of the argument is based on information provided by AUTOCERT's formal program analysis, and the argument structure follows the program and analysis structure, so that a largely automated safety case construction is possible. However, some information cannot be produced by the program analysis, such as environment constraints, external assumptions, list of related documents, or model names. This information must be specified externally by a safety engineer. This also applies to the formalization of the top-level safety requirements that drive AUTOCERT's analysis and their integration with the system-wide hazard analysis and safety case. Even though the constructed safety cases quickly become too large, an abstraction mechanisms can be used to highlight different aspects of the safety case. In particular, we can derive safety cases that are restricted to specific requirements, or to specific subsystems which is thus construct minimal but consistent safety case slices representing specific categories of information that help in manual safety case assessment.

In order to support the automated safety case construction, we integrate AUTOCERT's formal program analysis with an existing commercial safety case tool, Adelard's ASCE v3.5 tool [1]. We extended AUTOCERT to extract the manually specified

information from its own input and to structure this together with all information derived by the analysis into an XML format. The XML file records all the relevant information needed for the safety case construction. Subsequently, an XSLT program is used to transform this into a second XML format that logically represents the structure of the safety case as defined by safety case templates underlying the examples shown above. Here, the templates were designed so that the same argument structure can easily be adapted to other programs and systems. Finally, we use a custom Java program to present the safety case using GSN. The Java program helps to set the position of the nodes in the safety case which involved some mathematical calculations and to represent the argument to follow the standard Adelard ASCE file format. This architecture avoids a tight integration of the analysis (i.e., AUTOCERT) and presentation (i.e., ASCE) tools, and provides enough flexibility to change the latter with little effort.

The integration is largely completed; in particular, we have already fully automated the construction of the component-level safety cases that argue down to the code structure, and make up the overwhelming fraction of the combined safety case. However, the print quality of these large safety cases is insufficient for presentation, so we choose to recreate them in Microsoft Word here. The integration of system-level and component-level safety cases, as described in Section 3.5, requires further implementation work.

5 Related Work

The development and acceptance of a safety argument or safety case is a key element of safety regulation in most safety-critical sectors [19]. For example, Weaver [21] in his thesis presents arguments that reflect the contribution of software to a safety-critical system. Audsley et al. [4] present an argument based on correctness of the specification mapping, i.e., translation from the system specifications into a model and subsequently into code. Our work in contrast focuses on deriving a safety case that argues along the hierarchical structure of systems in model-based development and traces the safety requirements on the model via the inferred system structure to the code.

With the increased use of model-based development in safety-critical applications, the integration of safety cases into such approaches has become an important research topic. For example, Chen et al. [7] introduce an integration of model-based engineering with safety analysis and safety cases to help in assessing decisions in system design of automotive embedded systems. Hause and Thom [12] describe how SysML and UML can be used to model system requirements and how the safety requirements and other system elements identified in system design were used to construct the safety case. However, the focus in these papers is typically on extending various modelling frameworks to simply represent safety cases. Rushby [17] also uses automated theorem proving technology (based on the Yices SMT solver) to make a safety argument, but does not construct a detailed safety case. Moreover, his analysis starts with a manually constructed logic-based model of the system, whose connection to the underlying code remains unclear. In contrast, we focus on showing safety of the system on the code level and recover the slices of the system architecture to identify where in the system the safety requirements are established.

Most safety cases (see for example [11]) are constructed manually, as no advanced tools are available to support the automatic safety case construction. However, a manual

safety case construction [8] is far from satisfactory as it is a time-consuming and error-prone process. Most existing safety case construction tools only provide basic drawing support à la "boxes and arrows". For example, GSN: ASCE v3.5 from Adelard [1], the University of York Freeware Visio Add-on and GSNCaseMaker [2] are graphical tools for creating a safety case by means of a drag and drop interface based on a commercial drawing tool. Obviously, tools supported by automated analyzers such as AUTOCERT are needed to produce the complex safety arguments for software. In our work, we integrate formal analysis with a commercial safety case tool (i.e., Adelard's ASCE tool [1]) to automatically construct the safety case. Parallel to the work on safety cases described here, we have also used the same underlying information to create safety explanations in a textual form suitable for code reviews [10]. However, this work does not yet extend to the model-based reasoning level described here.

6 Conclusions and Future Work

We have described an approach where the hierarchical structure of systems in model-based development drives the construction of a hierarchical safety case for the generated code. Here, assurance is not implied by the trust in the generator but follows from a formal analysis of the code. The analysis is based on a set of formal safety requirements and provides formal proofs for use as evidence. We believe greater confidence in the assurance claims can be placed if the rationale behind the validity of the translation from the model to the program can be shown. We thus make explicit reference to the correct translation from the model level representation to the source level representation, including an argument over the formalization of the requirement. We show how the external assumptions on the systems input signals are used in establishing the safety of the program wrt. the given safety requirement. Like Rushby [17], we believe that "a safe design will have ensured that the assumptions are valid". Moreover, Littlewood et al. [15] explain why there is a very low probability of a claim that has been shown by a formal proof, actually being false, when the assumptions and evidence are valid. We thus believe that formal methods can provide the highest level of assurance when they are combined with explicit safety arguments such as the ones we derived here.

The work described here is still in progress, and we are currently completing the automatic construction of the safety cases. So far, we only consider nominal component behavior, but our approach could also be applied to the off-nominal case, provided that appropriate safety requirements for the off-nominal modes can be identified. We have applied our technique only to flight code generated by Real-Time Workshop from hierarchical Simulink models but we are confident that the same approach can be applied to other modelling systems and generators as well. Future work will focus on complementary safety cases that argue the safety of the certification framework itself, in particular the safety of the underlying safety logic (the language semantics and the safety policy). We believe that the result of our research will clearly communicate the safety claims, key safety requirements, and evidence required to trust the generated code.

Acknowledgements. This material is based upon work supported by NASA under awards NCC2-1426 and NNA07BB97C. The first author is funded by the Malaysian Government and USIM.

References

1. ASCE home page (2007), http://www.adelard.com/web/hnav/ASCE
2. CET GSNCase Maker (2007), http://www.esafetycase.com
3. Real-Time Workshop Embedded Coder (2007), http://www.mathworks.com/products/rtwembedded
4. Audsley, N.C., Bate, I.J., Crook-Dawkins, S.K.: Automatic Code Generation for Airborne Systems. In: IEEE Aerospace Conf., pp. 8–15. IEEE, Los Alamitos (2003)
5. Basir, N., Denney, E., Fischer, B.: Constructing a Safety Case for Automatically Generated Code from Formal Program Verification Information. In: Harrison, M.D., Sujan, M.-A. (eds.) SAFECOMP 2008. LNCS, vol. 5219, pp. 249–262. Springer, Heidelberg (2008)
6. Basir, N., Denney, E., Fischer, B.: Deriving Safety Cases from Automatically Constructed Proofs. In: 4th IET Intl. Conf. on System Safety (2009)
7. Chen, D.-J., Johansson, R., Lönn, H., Papadopoulos, Y., Sandberg, A., Törner, F., Törngren, M.: Modelling Support for Design of Safety-Critical Automotive Embedded Systems. In: Harrison, M.D., Sujan, M.-A. (eds.) SAFECOMP 2008. LNCS, vol. 5219, pp. 72–85. Springer, Heidelberg (2008)
8. Cockram, T., Lockwood, B.: Electronic Safety Cases: Challenges and Opportunities. In: Safety Critical Systems Symposium 2003. Springer, Heidelberg (2003)
9. Denney, E., Fischer, B.: A Generic Annotation Inference Algorithm for the Safety Certification of Automatically Generated Code. In: GPCE 2006, pp. 121–130. ACM, New York (2006)
10. Denney, E.: A Verification-Driven Approach to Traceability and Documentation for Auto-Generated Mathematical Software. In: ASE 2009, pp. 560–564. IEEE, Los Alamitos (2009)
11. Eurocontrol: Preliminary Safety Case for Enhanced Air Traffic Services in Non-Radar Areas using ADS-B Surveillance (2008)
12. Hause, M.C., Thom, F.: Integrated Safety Strategy to Model Driven Development with SysML. In: 2nd IET Intl. Conf. on System Safety, pp. 124–129 (2007)
13. Kelly, T.P.: Arguing Safety a Systematic Approach to Managing Safety Cases. PhD Thesis, University of York (1998)
14. Leveson, N.G., Cha, S.S., Shimeall, T.J.: Safety Verification of ADA Programs using Software Fault Trees. IEEE Software 8(4), 48–59 (1991)
15. Littlewood, B., Wright, D.: The Use of Multilegged Arguments to Increase Confidence in Safety Claims for Software-Based Systems: A Study Based on a BBN Analysis of an Idealized Example. IEEE Trans. Software Eng. 33(5), 347–365 (2007)
16. O'Halloran, C.: Model Based Code Verification. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 16–25. Springer, Heidelberg (2003)
17. Rushby, J.: A Safety-Case Approach For Certifying Adaptive Systems. In: AIAA Infotech@Aerospace Conference (2009).
18. Schloegel, K., Oglesby, D., Engstrom, E., Bhatt, D.: Composable Code Generation for Model-Based Development. In: Krall, A. (ed.) SCOPES 2003. LNCS, vol. 2826, pp. 211–225. Springer, Heidelberg (2003)
19. UK Ministry of Defence: 00-56 Safety Management Requirements for Defence Systems, Issue 4 (2007)
20. Vallado, D.A.: Fundamentals of Astrodynamics and Applications, 2nd edn. Microcosm Press and Kluwer Academic Publishers, Dordrecht (2001)
21. Weaver, R.A.: The Safety of Software-Constructing and Assuring Arguments. PhD Thesis, University of York (2003)
22. Weiss, K.A.: Component-Based Systems Engineering for Autonomous Spacecraft. MSc Thesis, Massachusetts Institute of Technology (2003)