Automated Component Configuration in Safety-Critical Domains

Zoë Stephenson[1] and John McDermid[1]

Department of Computer Science, University of York
Heslington, York YO10 5DD, UK

{zoe.stephenson, john.mcdermid}@cs.york.ac.uk

Abstract. Embedded systems development has enjoyed the success of product family technology for a number of years. However, the same success has not been present in the world of safety-critical embedded systems. These systems are developed using processes that fall under a great deal of scrutiny and justification, and automated tools to manage product family configurations will not be easy to accept in this type of process unless they exhibit some specific characteristics such as user control over processing and explicit traceability of processing steps. We propose an implementation framework for tools that are more amenable to this type of development process, and illustrate this framework with an application that configures fault-accommodation components for engine control software.

1 Introduction

Product-line technology has enjoyed a wave of successes [3] in a number of different contexts. It has been particularly successful when applied to small embedded system product lines — those situations in which a product line of embedding systems contain software that performs some of the functionality. There are a number of reasons for their success in these domains:

– The software requirements are based on the need to control a physical device or a protocol, rather than to respond to the needs of a human user. Device and protocol behaviours are traditionally easier to specify precisely than human behaviours.
– The number of variants of behaviour required of embedded software is generally limited by the range of variants in the embedding system.
– The required behaviour depends on very few different domains, and is overseen by relatively few stakeholders.

One particular class of embedded systems, however, has not enjoyed as much success with product-line technology — that of highly-integrated safety-critical systems. By 'highly-integrated', we mean that the parts of the physical system (in this case, an aircraft engine) work in unison to perform their function. These systems have some characteristics that differ from other embedded system applications:

– There is still a physical system that must be controlled, but the specification of device behaviour must include information about the way it can fail; the effects of those failures must be accounted for in the runtime operation of the system. This makes for more complex specifications.

- The number of variants is still very limited, but the dependencies that limit the range of choices are mostly based on the causal dependencies in the integrated embedding system, rather than in the dependencies of the functional behaviours. This increases the complexity of requirements derivation and product selection.
- The required behaviour is dependent on a larger number of domains, and the safety assessment procedures require a high degree of oversight from certification authorities. The structure of development deliverables, in particular, is mandated in certification standards. This places an additional constraint on product-line management technology, that it produces information for all of the intermediate deliverables required by the certification authorities, including traceability between those deliverables.

A large part of the responsibility of a safety-critical control system is to accommodate possible failures in the embedding system. This includes the various sensors and actuators, and for an aero-engine also includes subsystems such as the ignition system and the fuel pump. The strategies for dealing with failures in this system range from synthesising control parameters from redundant measurements, through less accurate reversionary control modes, to simply controlling the engine to a preset idle thrust.

The program logic that controls fault accommodation accounts for around 70% - 80% of the code for a modern aero-engine application. This includes code to detect the presence of faults as well as the fault accommodation behaviour. The requirements for this accommodation logic are derived from the safety assessment procedures, and are typically expressed in a highly formalised manner. Once the requirements are specified, the process of deriving an implementation is largely systematic.

A systematic transformation process that covers a large part of an application is a good candidate for automated translation. There is a large amount of effort involved in performing the transformation manually, and it is potentially applicable to a large proportion of the code, so the return on investment is likely to be high. Applying the technology to a product line of safety-critical systems brings an added benefit: any specific strategies that are developed so that the tool is more applicable to the domain are likely to be reusable across different products.

Tool development for safety-critical systems carries with it some additional concerns. For the aero-engine applications studied in this paper, those concerns are embodied in the DO178B standard[14]. In general, if a tool is to be completely trusted in operation, then its development must be controlled to the same standards as the development process for the product that it generates. For a general-purpose tool, this can become rather costly. An alternative strategy is to require that the tool's output is checked for correctness by a separate process. This additional process can be a separately-developed verification tool (for which less stringent criteria apply), a human review process, or a mixture of the two. However, this approach requires that the tool provide enough information about its operation to allow this review to take place.

This paper describes the design and implementation of a fault-accommodation generation tool for safety-critical aero-engine applications, using this alternative strategy of external verification. To ensure that the tools so developed report accurately on their activity, a general-purpose framework for externally-verified safety-critical transformation tools was generated. This framework supports the implementation of tools for product-line component generation and configuration under flexible user control.

2 Related Work

This type of transformational generation is typical of traditional compiler pipeline architectures. They parse input files into an internal representation and perform transformations on the internal representation to produce object code. For safety-critical software development, there is often a compromise between the transformational ability of the compiler and the need to verify that the object code correctly implements the source code. In the safety-critical domain, organisations are typically very conservative with preprocessors and compiler optimisations.

An application generator system raises the level of abstraction from that of source code, and allows the designer to work at a level that is closer to the domain in which the application is used. Early application generator systems grew out of a formal treatment of languages and transformational systems [2, 1]. Here, automation was seen as a way of guaranteeing that correct transformations were being applied to the product description.

A powerful and general system of transformations is found in the domain-specific application generator system Draco [12]. In this system, the transformation process is characterised as passing through a number of disparate but connected domains. Experts in one or more of these domains create transformation rules that manipulate the information within a domain, or translate information from one domain to another. The user of the system then directs the transformation process from one domain to another until an implementation domain is reached, from which the application may be compiled and executed.

Many of these generator systems have been used to create programs in narrow and well-understood domains of expertise. For example, the report describing Draco [12] uses the transformation of mathematical algorithms to illustrate the generator approach. The idea of focusing on a narrow domain of application can be seen in many domains where reuse has been successful [8], and in particular in modern product family research [5]. The typical example of a floating weather station product family [17] is a relatively small domain with user-focused variation. The use of larger and larger domains and more technological variation has only emerged in recent years as larger organisations take on product families [4].

An application generator system provides a type of reuse known as *generative* reuse. The generator and its input language encapsulate the knowledge that is being reused, and together they create artefacts that form part of the product. Some alternative strategies use *component* reuse — the artefacts that make up the product are already present in some kind of library repository[13], and they are extracted and integrated into the product according to its configuration, or the appropriate libraries are included with the product when it is distributed or executed.

The domain of engine control, the focus of this paper, is driven by system issues such as weight, emissions, safety and response times. It involves expertise in mechanical engineering, safety assessment, control engineering, aerodynamics and a whole range of associated areas. However, we believe that some subdomains, such as the accommodation behaviour, are sufficiently well-structured and formalised to warrant the use of some combination of generative and component-based reuse as a way of managing variability. An interesting characteristic of this scenario is that in the fault accommodation behaviour subdomain, it is the safety analysis process that provides the selection criteria for fault-accommodation products, rather than a customer-focused requirements analysis process.

3 Approach

The domain that has been chosen for the work presented in this paper is that of fault accommodation, with a safety analysis model providing selection criteria. There are some established techniques by which a safety model can be constructed [9]; these techniques describe the behaviour of the system in its normal mode of operation, the events that can cause abnormal operation, and the ability of the system to continue behaving correctly in the presence of those events. To continue correctly, some accommodation measures may need to be taken, and these are represented in the model so that they may be taken into account when assessing the safety of the system. In a typical engine control system, these measures are:

- For an intermittent or transient fault, the last good value can be latched to hide the problem until it is corrected
- If a value from a sensor is incorrect or unavailable, then a value from an alternative sensor may be used. Some sensors on the engines measure the same physical value as sensors on the aircraft, such as external pressure. For other values, there may be redundant sensors wired so that they reach the control computer by a different route. The engine control system can select among the values from these different sources to accommodate faults.
- If a sensor value is deemed to be incorrect, then it may be possible to calculate an approximation of that value from other sensor values. This will potentially lead to a small loss of accuracy in the control algorithms, but the effects of this loss of accuracy will have been analysed and accommodated in the design of the algorithms.
- If enough sensor inputs are unavailable, or if there is a problem with some actuators, then a different control algorithm may need to be selected. The system is still under control, but there may be a different control parameter in use (e.g. controlling based on a speed rather than a pressure) or the engine controller may only be able to control the engine to idle at a safe speed.

We seek to create an automated transformation tool for the generation of derived requirements and implementations from such specifications of required fault accommodation behaviour. The environment and process guidance within which the tool will operate imposes a number of tool implementation requirements. Some of these are to do with the reporting of the tool's actions, and others are to do with the ability to configure the tool to use particular strategies for particular inputs. Key requirements are:

1. The tool shall be made of data processing activities that incrementally convert input data to output data.

2. The selection of data processing activities, where more than one is applicable, shall be under the control of the tool user.
3. Each data processing activity shall be independently testable.
4. Each data processing activity shall report on the context in which it was invoked, the output that it produced, and the alternative rules that the user could have selected among.
5. Each invocation of the tool shall be tagged with a unique identifier and configuration management information (user, date, time, revision etc.) as appropriate to the project's configuration management criteria.

Together, these requirements ensure that the use of the tool will be as transparent as possible. This helps to ensure that automated or manual review processes have access to as much information as possible about the tool operation, and that when problems arise (such as accommodation measures that are intrinsic to the control loops, which cannot be automated in this way) there is the possibility of selectively overriding the default tool operation to correct the problem. The division of tool processing into incremental processing activities means that if there is currently no strategy in place to handle a particular problem, one can easily be added.

These requirements are realised in an application framework. The architecture of the framework is depicted in Figure 1. The framework allows for the specification of a number of transformational processes. Processes are connected to data repositories, and do not directly communicate with one another. The program's input comes from input-facing processes that take external information and build internal representations; conversely, output-facing processes take internal representations of transformed information and provide output to external information.

Each individual transformation process is described as a tuple containing the source, sink and name of the process. Separately to this is a user-control manager system. This records matching expressions that specify a process and input names and values, and allows the user to divert the execution into a different process. This constrains the range of data types that can be stored in the intermediate data areas, as the user control input language must be able to specify matching expressions (e.g. "value > 4", or "name != 'CriticalSection' "). However, the loss of expressiveness here is offset by the gain in control over tool behaviour, which is crucial for the type of development process for which this tool framework is intended.

After each processing element performs its transformation, that transformation is recorded in a database of traceability information. The record contains the name of the original process that was selected, the rule that was actually executed, all of the input values supplied and all of the output values produced. This provides a complete log of the tool activity throughout the transformation process. Information of this kind is needed so that it can be shown that the tool has either correctly selected the appropriate transformation or has correctly obeyed the user control information. The same data can also be used for requirements traceability databases and rich traceability arguments.

The fault-accommodation generation tool is implemented as two separate applications of this framework, one to derive human-readable requirements from the safety analysis information and another to generate the component configuration required to meet those requirements. They are divided so that the user can review the derived re-

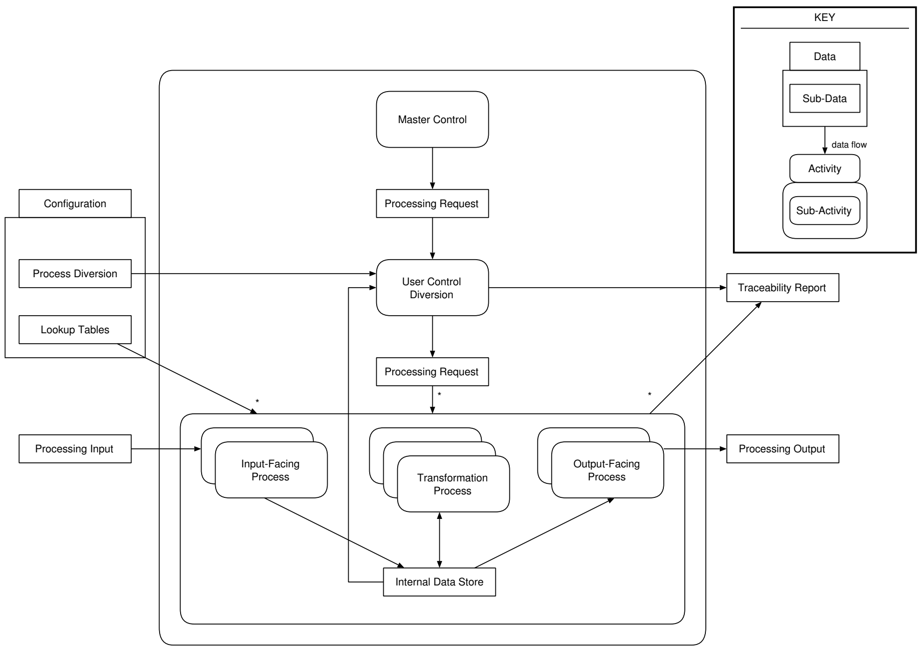

Fig. 1. Tool Framework Architecture

quirements before the implementation is configured. This stage may result in changes to the user control information used in generating the derived requirements, iterating until an acceptable set of requirements is reached. The user may also add new requirements from outside the derivation tool, and mask out requirements for which automated configuration is not desirable. The application framework organises transformations as a number of small processes chained together, and manages the data interactions outside of those processes. Even though the framework allows transformations to occur as soon as there is data available, the application uses a strict pipeline approach to preserve determinism and improve the readability of processing traceability output. This makes it very straightforward to divide a transformations into a number of discrete stages so that the user may intervene and review progress between stages. This first derivation stage is an example of generative reuse based on the safety analysis information.

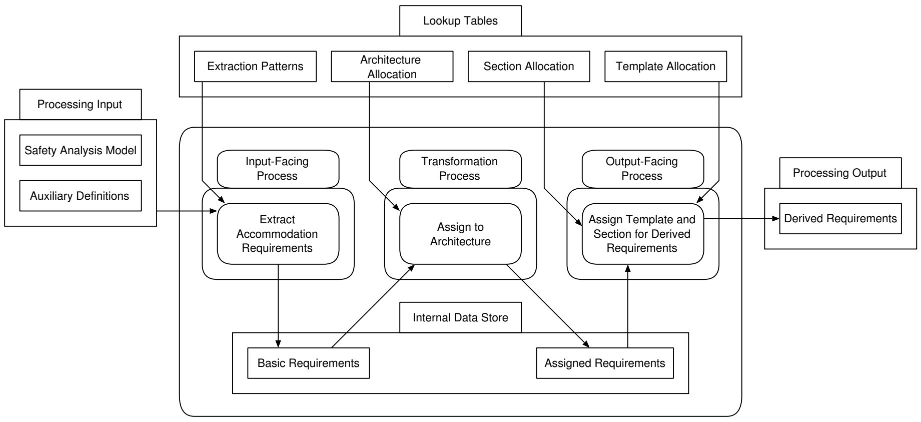

Fig. 2. Requirements Allocation Tool

Requirements derivation and allocation is shown in Figure 2. This stage of the tool is concerned with extracting the required fault accommodation from the output of the safety analysis process. Extraction is performed by pattern-matching the different parts of the safety model against a set of patterns representing those parts of the model that require accommodation. The process combines these requirements with auxiliary information that specifies how to perform accommodation in more detail, such as cross-check tolerances and mode signals, that is not present in the abstract safety model. The tool accounts for the operation by attaching a set of modes to each requirement. If any modes are attached, then the requirement is only valid in those modes. The requirements are allocated to specific elements of the product-line software architecture and mapped to the project's documentation structure through pattern-matching tables. The resulting requirements are output in both machine-readable and human-readable forms using a set of customisable templates. The user is able to control the allocation of templates to requirements at this stage. This allows for specific types of formatting or phrasing, such as the use of truth-tables or structured language requirements.

This stage of the tool is an example of generative reuse based on the safety analysis information.

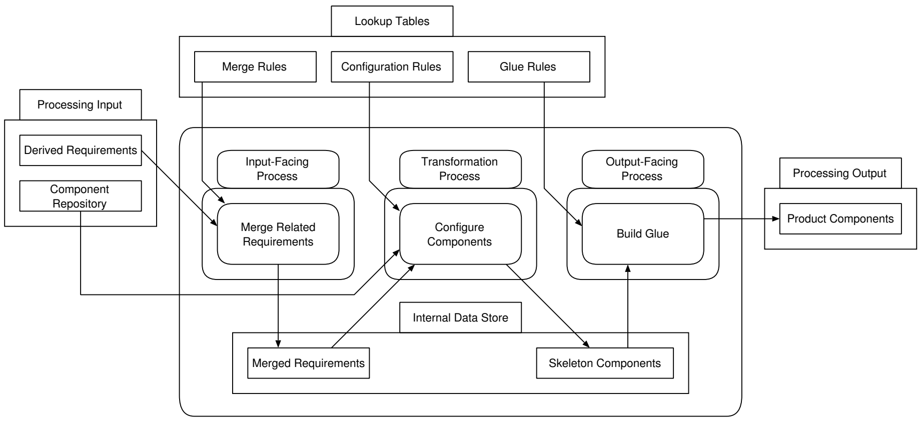

Fig. 3. Implementation Configuration Tool

The implementation configuration stage is shown in Figure 3. This tool is concerned with the organisation and interconnection of existing library components that have been produced for use within a product-line architecture. The accommodation processing requirements from the first stage of the tool specify bindings for their inputs and outputs. The configuration tool links the individual processing specifications together according to their bindings, to form complete processing transactions. The bindings are only considered at this stage if both of the components involved are allocated to the same architectural unit. The transaction grouping process provides all of the information needed to configure a set of components for the accommodation transactions. The components are created according to their architectural units in a combination of the following ways:

1. Certain patterns of required accommodation behaviour are mapped to existing product-line components by pattern-matching against a table of existing patterns and the corresponding components. For example, a range check and a cross check in series might map to a particular configuration of a combined range and cross checking component.
2. Certain patterns of required accommodation behaviour are created by instantiating simple operators into component wrappers. For example, an existing accommodation component can be made mode-specific by wrapping it with a switch that only routes processing through that component when the system is in an appropriate mode.

This pattern-matching process is driven by another configurable table that maps between patterns of requirements and the names of the corresponding transformation to use. That name is then checked against the global user control database to allow the user to override the rule that has been selected.

The process results in a group of configured components for each architectural unit within the product-line software architecture. Within each group of components, some of the requirements may have been merged to correspond with the set of reusable components provided within the product line. This merging is reported through the requirements traceability information, so that it may be dealt with appropriately when constructing traceability links or traceability matrixes.

This stage of the tool is mainly component-based reuse, with a trivial amount of generation added to glue components together and impose mode-specific behaviour.

4 Evaluation

Evaluation is an important process for any tool. For a tool that is intended to produce safety-critical implementations, the evaluation and assessment process is critical to the use of the tool. The evaluation described here is our first step in assuring that the tool is able to correctly transform safety analysis information into the appropriate accommodation components.

To provide this assurance, the tool is exercised on safety analysis information from an existing project, generating derived requirements in a form that is appropriate to the project. This includes the particular requirements documentation structure and the templating of individual requirements. The derived requirements must make reference to the units of the existing product family architecture. When the implementation is generated, the components that are configured must be of a comparable granularity and responsibility to those used in the original product.

The evaluation compares the results of automated generation of fault accommodation implementations with the results obtained in the original project by software engineers constructing the design manually. The features that are of interest during the comparison are:

- Fault accommodation functionality that was specified in the manually derived requirements but was not produced through automated generation.
- Fault accommodation functionality produced through automated generation that was not specified in the manually derived requirements.
- Functionality that is structured differently when specified through automated generation than the manually created functionality.
- Implementation features in the manually-derived fault-accommodation system that are not present in the automatically generated system.
- Implementation features in the automatically generated fault-accommodation system that are not present in the manually-derived system.

An issue with this evaluation is the level of user control used during evaluation. For some of the features that are generated automatically, there will be a number of different ways of expressing the accommodation requirement and of configuring the implementation. If the automated tool were being used by the author of the original derived requirements and the author of the original software, then they would use the user control facilities to ensure that certain choices were made, and those choices would cause the requirements and implementation to be expressed in a way that they find appropriate. The evaluation should take into account at least the behaviour of the tool without user control and with user control that models the decisions that were made when the original derived requirements were produced.

The potential need to alter the tool behaviour does have an advantage, however. If a reason can be attributed to the user control specifications, this would help in validating the rationale behind the decisions that were made in the original project.

5 Conclusions and Further Work

As of the time of writing, the tool framework has been created and the tool has been exercised on some small example specifications to validate the ability to control selection between different transformation processes. However, a full implementation of the requirements allocation tool is still under way. The tool is currently targeted at a safety analysis modelling notation that is unique to Rolls-Royce and is manipulated using custom tools; some additional effort is being put into the tool development to ensure that the interpretation of the safety modelling information can be retargeted to commercial safety analysis toolsets such as the Item toolkit[15] or FaultTree+ [7].

Product-line techniques as a whole are well-known for reducing defects and improving software quality when they are used effectively. However, for an organisation that is in the process of moving to a product-line technique, the investment in assets and cultural change is a huge risk, especially when the operation of the business is constrained by the certification standards for their domain of operation. A configuration management tool such as that described here represents a lower-risk approach that can help the organisation to migrate incrementally to a product-line approach.

The work presented in this paper outlines an automation approach to the configuration of assets, but does not describe any additional support for the construction of those assets. One interesting avenue of further work would be to determine any impact that the structure of those assets and their architecture has on the ability of the designer to understand and manage the resulting configuration. This focus on automating the routine steps of product configuration in this small domain helps to ensure that variation in this domain is managed at a higher level of abstraction and at an earlier point in the lifecycle. More effort can be spent on defining and validating safe product configurations that respond to changing requirements and physical system descriptions throughout product development. This will mean that variability mechanisms are introduced at the level of the integrated system, rather than just at the level of software; there is some work that has been performed in this area to apply ideas of product family variation to this domain [11, 10, 6, 16].

One advantage of automated transformation that was touched on earlier in this paper was the potential for automated traceability between safety models, requirements and implementations. If the tool is being customised in a particular way for a particular product, then rationale can be added to that customisation to describe the reason for a transformation step to be carried out in a certain way. All of this information can be used to improve and validate traceability links between design artefacts.

References

[1] R. Balzer. A 15 Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, November 1985.

[2] R. Balzer, T. E. Cheatham, and C. Green. Software Technology in the 1990s: Using a New Paradigm. *Computer*, 16(11):39–45, November 1983.

[3] J. Bosch. Product-line Architectures in Industry: A Case Study. In *Proceedings of the 21st International Conference on Software Engineering*, pages 544–554, May 1999.

[4] J. Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.

[5] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6):37–45, November 1998.

[6] J. Dehlinger and R. R. Lutz. Software Fault Tree Analysis for Product Lines. In *IEEE International Symposium on High Assurance Systems Engineering*, 2004.

[7] Isograph. Isograph Reliability and Safety Software Products. http://www.isograph-software.com/.

[8] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[9] N. G. Leveson. *Safeware: System Safety and Computers*. Addisson-Wesley, 1995.

[10] D. Lu and R. R. Lutz. Fault Contribution Trees for Product Families. In *International Symposium on Software Reliability Engineering*, pages 231–242, November 2002.

[11] R. R. Lutz. Toward Safe Reuse of Product Family Specifications. In *Proceedings of the Fifth Symposium on Software Reusability*, pages 17–26, May 1999.

[12] J. M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, 10(5):564–574, September 1984.

[13] R. Prieto-Díaz. Domain Analysis: An Introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, April 1990.

[14] RTCA. Software Considerations in Airborne Systems and Equipment Certification. Technical Report 178B, Requirements and Technical Concepts for Aviation, 1992.

[15] Item Software. Item Toolkit. http://www.itemsoft.com/.

[16] Z. R. Stephenson, S. de Souza, and J. A McDermid. Product Line Analysis and the System Safety Process. In *Proceedings of the 22nd International System Safety Conference*, August 2004.

[17] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Development Process*. Addison-Wesley, 1999.