Assurance Cases for Block-configurable Software

Richard Hawkins[1], Alvaro Miyazawa[1], Ana Cavalcanti[1], Tim Kelly[1], and John Rowlands[2]

[1] Department of Computer Science, University of York, York, UK
[2] BAE Systems, Warton Aerodrome, Preston, PR4 1AX, UK

Abstract. One means of supporting software evolution is to adopt an architecture where the function of the software is defined through reconfiguring the flow of execution and parameters of pre-existing components. For such software it is desirable to maximise the reuse of assurance assets, and minimise re-verification effort in the presence of change. In this paper we describe how a modular assurance case can be established based upon formal analysis of the necessary preconditions of the component. Our approach supports the reuse of arguments and evidence established for components, including the results of the formal analysis.

1 Introduction

Software maintenance and evolution is typically very costly. In the safety-critical domain, extensibility and reconfigurability have to be traded for simplicity, with impact on maintainability. We consider here what we call block-configurable software, which achieves this compromise by adopting an architecture that supports configuration via structured input data.

Block-configurable software comprises a number of components that provide particular functionality, and a manager, which uses configuration data to define how the components cooperate. The architecture resembles that of a control-law diagram with connections defined by configuration data. Block-configurable software is a convenient means of implementing control systems for which changes in dynamic behaviour can be restricted to changes in the parameters and connections of a fixed set of components.

Block-configurable software facilitates changes: to add or to change a function, it may be not be necessary to touch the code at all. It is also easier for a third-party to implement changes, since it may be enough simply to provide appropriate data. So, the integrity of the code can be maintained, whilst flexibility is still provided to the user.

This gives rise, however, to a challenge for assurance. To realise the benefits of block-configurable software, it is necessary to limit the work required for validation in the face of changes. Also from an assurance perspective, the impact should be limited to the configuration data.

The validity of the configuration data provided is a key aspect of the assurance case. We need to identify the constraints that characterise valid data, and to consider the way in which validity is established. This can be related to concerns regarding exceptional behaviour, use of resources, or any other general properties. To address both the identification and verification of constraints on the data we adopt the use of formal analysis.

Our contribution is a general pattern for assurance cases that can be made for block-configurable software using a combination of formal analysis and more traditional verification. Safety-argument patterns provide a way of documenting and reusing argument structures by abstracting the fundamental strategies from the details of a particular argument. It is possible to create specific arguments by instantiating the patterns in a manner appropriate to the application. We present a number of options for supporting various aspects of the assurance cases. We also consider the effect that changes to the software have on the assurance case, and how the impact can be minimised. We maximise reusability both at the level of the structural arguments and of the formal analysis.

Our assurance cases are not for particular properties of the system; they demonstrate that the software does not adversely affect the system in which it is embedded. We have applied our approach in an industrial case study, but use a quadratic-equation solver as an example here.

To represent assurance arguments clearly, we use a graphical notation: the Goal Structuring Notation (GSN), as it is mature, widely used, and standardised [6]. We observe, however, that our results apply to any notation that conforms to the meta-model of the OMG standard for structured assurance cases (including Adelard's CAE, for example) [12].

Section 2 defines the characteristics of block-configurable software. Our assurance-case pattern is described in Section 3. Section 4 discusses the well-behavedness arguments, and in particular termination. Section 5 discusses the effects of likely changes. Section 6 explains how the validity of configuration data can be established using a formal approach. Finally, Section 7 discusses related work and Section 8 provides conclusions.

2 Block-configurable software

Block-configurable software is created from generic components. It may be used to implement a solution for any problem that requires a vector of inputs to be transformed to a vector of output values. The functionality of the block-configurable software is determined at runtime through the connection of the components and the provision of parameters to them as

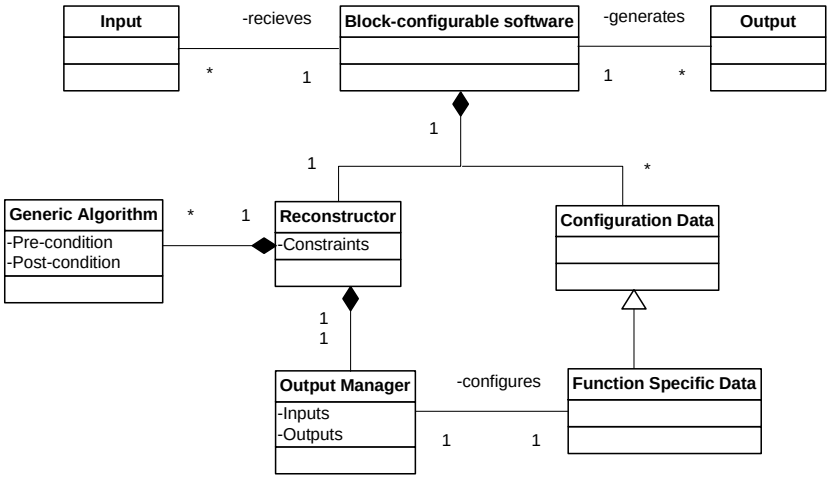

Fig. 1. Structure of block-configurable software

defined using loaded configuration data specific to a particular function. A unique characteristic of this type of software, in comparison for example to data-driven software, is that data is also the means by which software is configured at design-time, through defining parameters for the code blocks and connections between them.

A key feature of block-configurable software is that it is extensible; it allows the user to employ any number of components, which can be used in any sequence to derive the required outputs. The configuration data provides all the information required to define the inputs that are needed, the outputs that are to be generated from the inputs using the components, and the parameters that are given to each component.

Figure 1 shows the structure of block-configurable software. **Inputs** and **Outputs** characterise its interface. A **Reconstructor** characterises the functions provided by the software using one or more generic components. **Configuration Data** for a function is selected at runtime. **Function Specific Data** configures the software for a particular function. **The Output Manager** is a component that defines the parameters provided to the other components and their order of execution. Each function must include an output manager, and it must be the first component executed (to define how the others are used). The design of the reconstructor and the output manager are the same in all block-configurable software. The only changes ever required to the reconstructor are small adaptations to take into account the introduction and removal of components.

A deployment of block-configurable software may include a number of different functions. Each function is reconstructed using a subset of the components available and function-specific configuration data. A deployment, therefore, consists of a set of generic components, a reconstructor that can reference all of them, and configuration data for each function. New functions can be added to the deployment by adding new configuration data, as long as only the existing components are required.

As an example, we consider a quadratic-equation solver; its Ada implementation is presented in [13]. Its inputs are the coefficients of a quadratic equation and it generates as output its solutions. The configuration data can be used to select one or both of them as output. The reconstructor uses generic components called `OUTPUT_MANAGER`, `ADDER`, `MULTIPLIER`, `SQRT`, and so on. The software may, therefore, implement different functions that uses addition, and square root, for example.

3 An assurance case for block-configurable software

In defining the structure for the assurance arguments of block-configurable software, there have been two primary considerations: resilience to expected change scenarios and creation of libraries of assurance-case modules for generic components and function-specific configuration data. Figure 2 shows the structure of the assurance-case pattern we have defined.

Figure 3 shows the argument within the top Block-Configurable Software module, which supports the overall claim that the block-configurable software does not adversely affect the system. To demonstrate this, we show that the software itself does not adversely affect the system and all the constraints are met by the configuration data for that application.

The claims relating to the components and the configuration data are in separate modules. This allows the demonstration that the configuration data meets the constraints to be done independently from the analysis of the components. The connection between the two parts of the argument is the constraints, which are derived as part of the generic-components argument and used by the configuration-data argument. In Figure 3 this is captured by Away Context definitions (boxes with rounded top) associated with the Away Goal regarding the data constraints. The Away Goal is defined in the module Configuration Data (as named in the bottom of the Away Goal symbol). The Away Contexts are defined in the Generic-Component module (also named at the bottom).

Our approach relies on the use of the block-configurable software pattern. The assurance argument must, therefore, include evidence that the

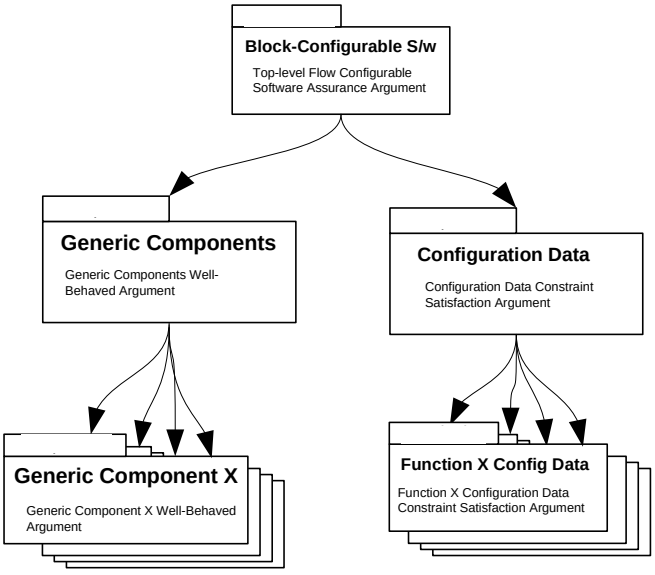

Fig. 2. Assurance-Case Architecture

software has the necessary characteristics identified by this pattern. This can be demonstrated, for example, through a simple manual check of the structure of the software modules against the pattern.

The argument regarding the generic components demonstrates that the software is "well behaved". In the argument, we assert that "if well behavedness is demonstrated, this ensures that the software does not adversely affect the system". Notions of interest for "well behavedness" are related to termination, resources, and exceptions. Here, we only consider the argument made for termination. In [13], we explore other arguments. A confidence argument is required to show that the identification of concerns regarding well behavedness is complete and correct. The argument considers each of these concerns in turn.

The argument regarding termination must be valid for all functions of the deployment. As shown in the instantiation of our pattern for the quadratic solver example in Figure 4, the argument considers the termination guarantees of the reconstructor as well as those of each of the generic components. The claim that must be demonstrated is that the guarantee of termination is achieved for each component if the defined constraints on the configuration data are met.

An argument module (Generic Component X) is created for each of the generic components and for the reconstructor. This allows the argu-

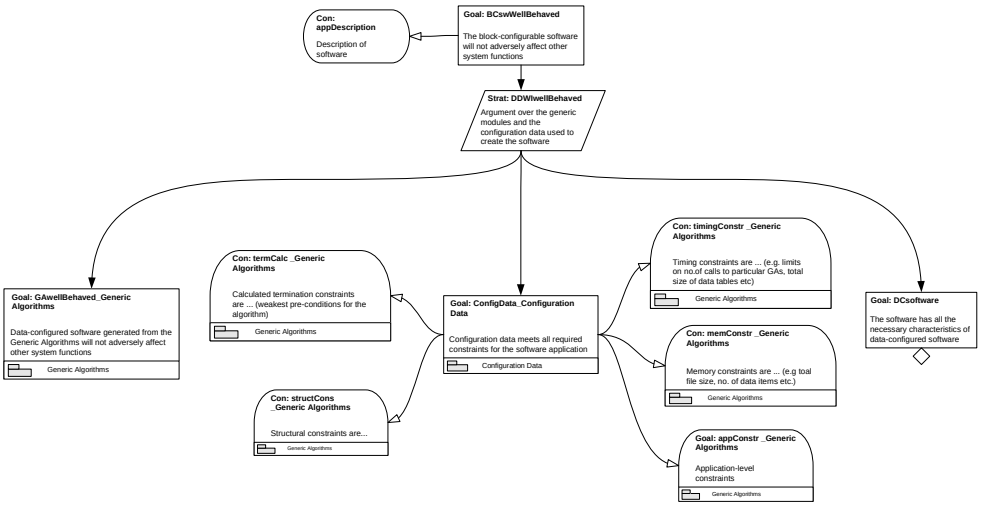

Fig. 3. Assurance Argument: BC Software Module

ment for each component to be reused for different functions. The reconstructor may require small changes for different deployments. A generic component, on the other hand, may be reused, as is, for any deployment. Since it is expected that the generic components are used across different applications, this can provide a large saving in reverification effort.

We omit the pattern for the Configuration Data module; it can be found in [13]. Its argument establishes that the configuration data meets all of the constraints determined within the components argument.

We have a separate argument for each function, with the assumption that only one function executes at any time (otherwise combinations of functions need to be considered). This argument structure creates assurance components for function-specific configuration data, and facilitates reuse for particular functions across multiple deployments.

We envisage three strategies that can be adopted to show that the constraints are upheld by the configuration data. First, if the constraints are simple, manual review is possible. This is easy to implement and requires no specialist tools or techniques. It is, however, infeasible for more complex data and constraints, and provides low assurance.

Rather than checking the configuration data, it is possible to consider the process for its generation. A systematic process may begin with an abstract representation of the function (such as a data-flow diagram including the relevant generic components). This abstract model can be

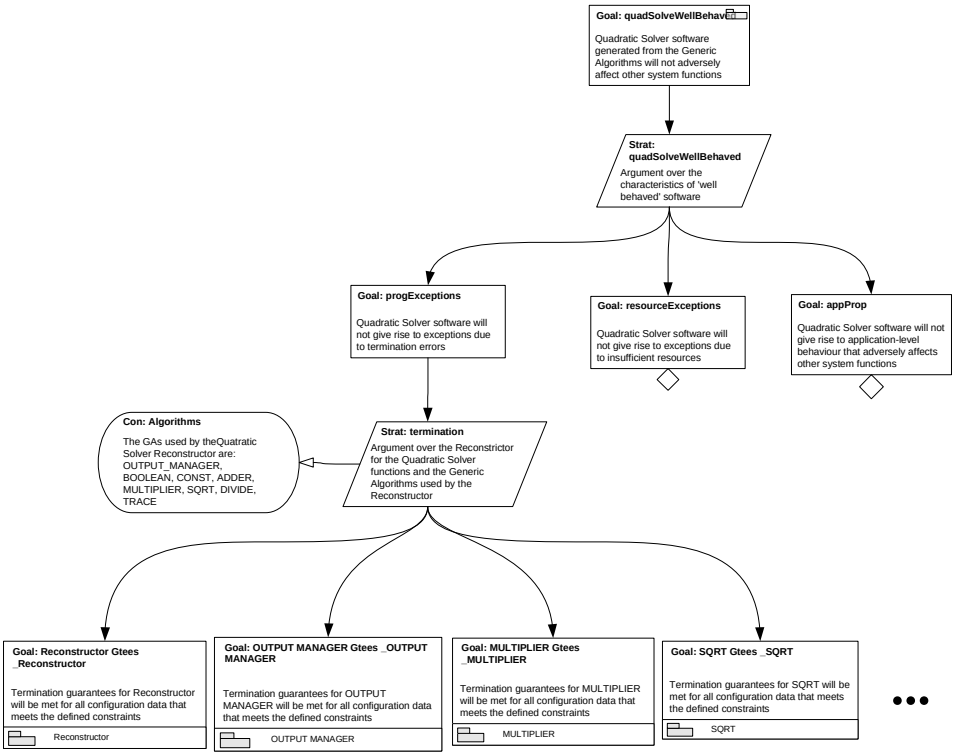

Fig. 4. Quadratic-Solver Assurance Argument: Components Module

verified to check that it is structured to ensure essential properties, like the correct number of parameters are defined for each block. This can also be an effective method of establishing application-level constraints, since it provides a view of the required inputs and outputs as well as an end-to-end view of the components used. The process of transforming that abstract model into configuration data can also be used to enforce constraints on the data. This requires reliable (probably bespoke) tool support and correct encoding of the constraints within the tool.

Finally, it is possible to prove formally that the configuration data satisfies the constraints using SAT (Satisfiability) or SMT (Satisfiability Modulo Theories) solvers. There are a number of tools available that implement such techniques [7]. This has the potential to give the highest available level of assurance. Its feasibility, however, depends on the structure of both the constraints and the configuration data.

4 Termination

In this section, we consider the arguments for well behavedness as defined in the Generic Component X argument module (Figure 5). It illustrates our approach to combining structured argumentation and formal analysis.

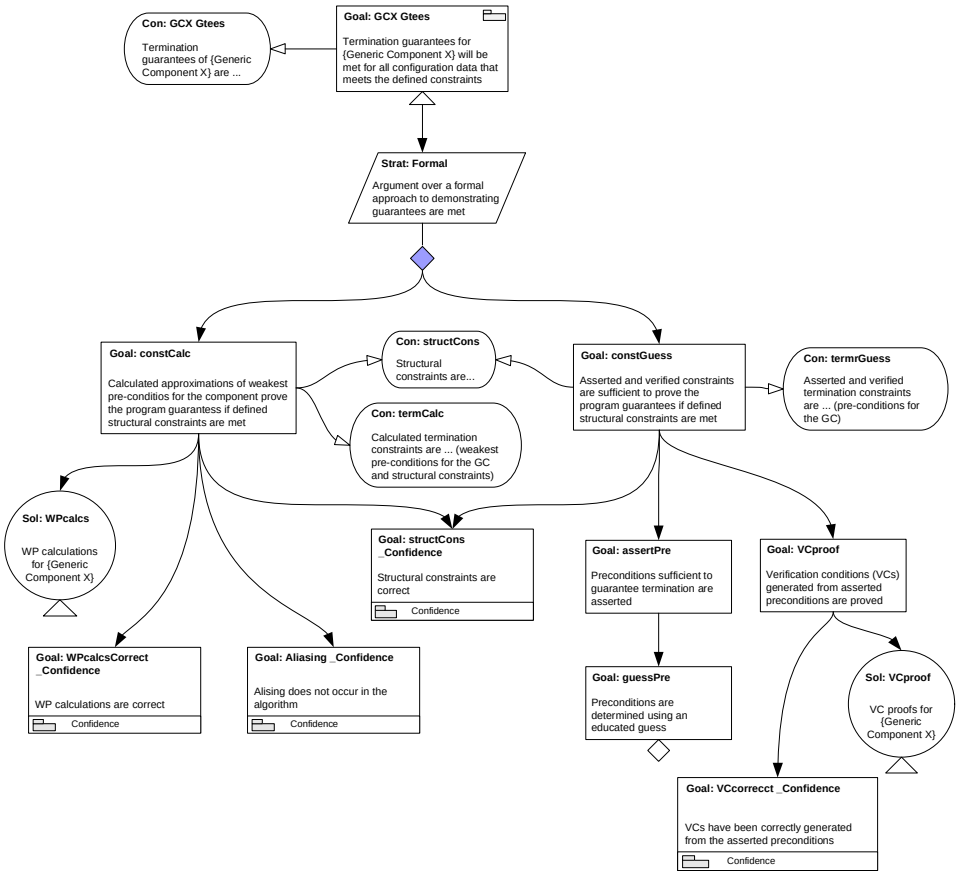

Fig. 5. Assurance Argument: Generic Component X

In arguing termination, a possible approach is to test the components. This requires test cases that provide sufficient coverage of the configuration data. Without unrealistically constraining the configuration data, however, this is extremely difficult. It is possible to use typical data, but extrapolating those test results would not provide much confidence. An

alternative would be to test the components every time new configuration data is used. This, however, does not permit reuse of evidence.

In addition, testing gives no indication of the constraints that need to be satisfied, and so no guidance for the definition of configuration data. The constraints can serve as a contract between the software developers and those configuring the software to provide particular functionality.

An alternative is to use formal analysis to prove that the termination guarantees are always met. There are two possibilities: to calculate weakest preconditions that guarantee termination of the components, or assert constraints that are believed to be sufficient and then verify that they guarantee termination. The asserted preconditions can be obtained through an educated guess, based on an understanding of the code.

There are advantages and disadvantages to each of these possibilities. The advantage of formal analysis is that it gives the highest available level of assurance, as it is based on proof. The disadvantage is that currently no tool support exists for calculating the weakest preconditions of Ada programs. We note, however, that due to the structure of the assurance argument, the calculations for each generic algorithm only needs to be performed once, and after that can be reused for all functions requiring that algorithm. The main advantage of the guess-and-verify approach is that there is an existing tool set available - the SPARK toolset [21] - that can generate and prove verification conditions. The disadvantage is that guessing the precondition may be difficult, and there is no guarantee that the correct precondition will be found. Based on this, we have decided to adopt a weakest precondition approach.

Weakest preconditions [8, 9] can be calculated using a function $\mathcal{WP}.\texttt{P}.\psi$ that defines, for a given program `P` and postcondition ψ, the weakest precondition ϕ that guarantees that, if `P` is executed in a state that satisfies ϕ, then it terminates and the final state satisfies ψ. The predicates ϕ and ψ establish restrictions on the values of the programming variables, and in our case, the preconditions are restrictions on the configuration data imposed by the implementation. (More details are provided in Section 6.)

Confidence arguments are required to demonstrate that constraints are complete and correct. An example is provided in [13] of a confidence argument to demonstrate the correctness of the weakest precondition calculations. This ensures completeness as well.

In Figure 6, we present the argument in the reconstructor module, which instantiates the pattern in Figure 5. The instantiation is guided by the adoption of a weakest precondition approach. There are confidence arguments that must also be provided. The argument for the reconstruc-

tor is similar in all block-configurable software. The similar argument for a specific component is provided in [13]. The argument for the output manager is exactly the same for all block-configurable software.

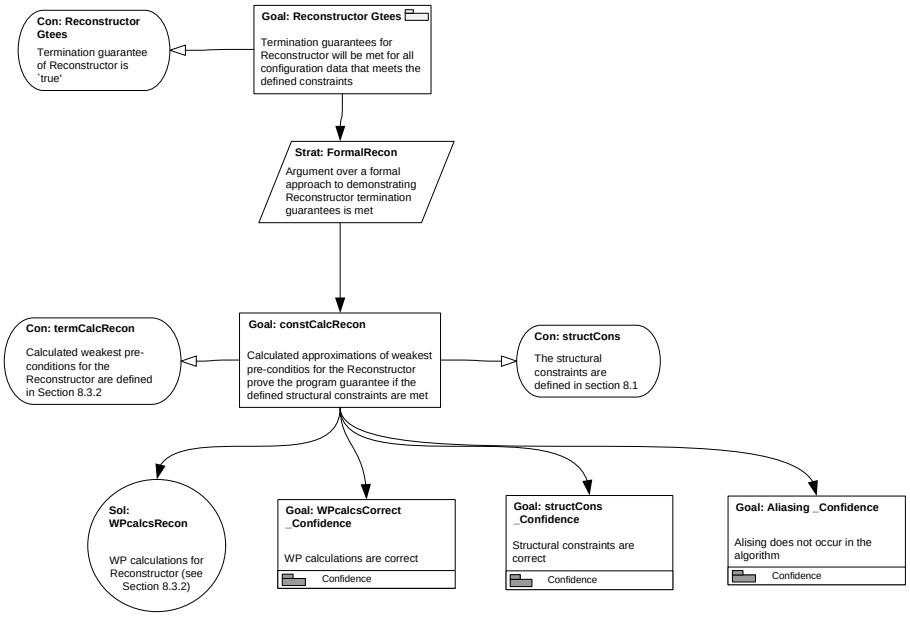

Fig. 6. Quadratic-Solver Assurance Argument: Termination Argument

As shown in Figure 3, the top-level argument requires a Configuration Data module that demonstrates that the constraints are met by all the configuration data. An argument module needs to be created for each set of configuration data. For the quadratic example, we assume that two different sets of configuration data are used by the reconstructor, which, for instance, require a different subset of the equation solutions.

5 Managing changes to block-configurable software

We consider three of the most likely change scenarios for block-configurable software, and discuss their effect on the assurance argument and evidence.

Add a new function to a deployment In this scenario, we assume that the new reconstructor only requires the use of the existing generic components. In this case, another Function X Configuration Data module must

be created for the new configuration data, and the Configuration Data module must be updated with an additional away goal reference to it.

Modify configuration data for an existing function Here, the behaviour of a function needs to be modified, but there are no required changes to generic components. In this case, only the existing Function X Configuration Data module for the changed function must be updated.

Introduce a new generic component Now, a new function or a change to an existing function requires a new generic component, and, correspondingly, a new Generic Component X argument module. This can generate new constraints, and so, the Function X Configured Data argument modules need to demonstrate that all constraints, including the new constraints, are met. In addition, due to required changes to the reconstructor, its Generic Component X argument module needs to be reassessed.

In conclusion, the best possible outcome is if, as a result of changes, all that is required is a reverification of data constraints. For that, it must be possible to demonstrate the required guarantees purely through data constraints. In addition, these required guarantees must remain unchanged. In practice, this is possible only in restricted scenarios of change. It is crucial, however, that the argument considers each generic component independently. It is also necessary to be able to make generic guarantees for the components in the absence of specific data.

Changes to the structure of the reconstructor (that is, those not implemented through configuration data), should be avoided. If object-oriented features can be used in a particular rendering of a block-configurable software, this becomes easier. In this case, the reconstructor can be implemented (and assured) once and for all, in terms of an abstract class that captures a generic interface for a component.

6 Validity of configuration data

There are different sources and kinds of constraints. Structural constraints derive from the way the block-configurable software is designed. These constraints are mostly independent of the particular application for which the software is used (except in the definition of the particular data types used in the generic components).

We have formalised the structural constraints using Z [18, 1]. Our Z data model is the same for all deployments of block-configurable software, except only for the data types of the generic components. Changes are,

therefore, only needed if different components are considered, and, in any case, the modelling effort required is small, since the data types are just records that can be directly represented in Z.

Our model ignores the use of pointers. For the quadratic solver, we have an Ada implementation where access types are used just to allow the use of unconstrained array types in a way that ensures absence of aliasing. In general, we need a technique to ensure that aliasing does not occur. This is addressed by a structured argument as shown in Figure 5.

To illustrate how we model the data types of generic components, we show below the model for the `ADDER` component of the quadratic solver. It is a straightforward translation of the Ada code to Z. The `DATA_PACKET_TYPE` is a record (schema) that includes the parameter and the components of the Ada record. The predicate (invariant) of the schema defines the range restriction in the declaration of the type.

$$
\begin{array}{|l}
\hline
RECONSTRUCTOR\_ADDER\_DATA\_PACKET\_TYPE \\
INPUT\_SIZE : POSITIVE\_WORD\_TYPE \\
INPUT\_SCALING : PARAMETER\_ARRAY\_TYPE \\
\hline
\#\, INPUT\_SCALING = INPUT\_SIZE \\
\hline
\end{array}
$$

Other constants, which define inputs and an identifier for the generic component, are also defined by a direct translation of the code.

More interesting is the model of the `OUTPUT_MANAGER`, which embodies the structures that allow the data configuration. This model, except in its dependency on the definitions of the $DATA\_PACKET\_TYPE$ schemas for each of the generic components, is the same for all deployments. This is the most complex part of the model, but since the output manager is a generic component, what we have is a $DATA\_PACKET\_TYPE$ record.

$$
\begin{array}{|l}
\hline
RECONSTRUCTOR\_O\_M\_DATA\_PACKET\_TYPE \\
OUTPUT\_MANAGER\_DATA : O\_M\_DATA\_TYPE \\
\hline
\end{array}
$$

The $O\_M\_DATA\_TYPE$, however, is a record with six components that define the size of the output vector, the vector of outputs actually provided and its size, and the dependencies between the outputs and inputs. The Z model for it and all the quadratic solver components is in [13].

The configuration data is an array of $DATA\_PACKET\_TYPE$ records.

$$
\begin{array}{l}
CONFIGURATION\_DATA\_TYPE == \\
\quad ARRAY[POSITIVE\_WORD\_TYPE, \\
\qquad RECONSTRUCTOR\_DATA\_PACKET\_TYPE]
\end{array}
$$

The instantiation of this model for a particular deployment defines the type *RECONSTRUCTOR_DATA_PACKET_TYPE*, which aggregates the possible *DATA_PACKET_TYPE* records used in the components. We calculate the weakest precondition of the reconstructor, which restricts the values of a record *CONTRACT* that includes a component of type *CONFIGURATION_DATA_TYPE*, and two others to represent the input and a selection of outputs.

There may also be domain constraints on inputs that arise in the area of application. They typically restrict the range of the values of the inputs (for example, height, speed, and so on). For our example, we require that the first coefficient of the equation is different from 0, otherwise it is not a quadratic equation. It is necessary to prove that any configuration data to be used satisfies the programming and structural constraints. For that, domain constraints can be assumed to hold. This is demonstrated in the Configuration Data argument module.

To calculate weakest preconditions, we define the function $\mathcal{WP}(S).\psi$, and consider the postcondition **True**. The definition of $\mathcal{WP}(S).\psi$ is mostly standard, except that we consider that expressions can raise exceptions and prevent proper termination. We, therefore, use auxiliary functions that determine when a expression or a command terminates. Definitions are provided in [13], along with calculations for the quadratic solver.

A significant part of those calculations, namely, the treatment of the reconstructor and the output manager, is reusable, and does not need to be revisited for other software. In addition, as long as the components terminate, the calculation of the weakest precondition is compositional: if a component is added, it can be considered in isolation, and no recalculation is needed. This is despite the fact that the components can be enlisted by the reconstructor in any order.

7 Related work

As far as we know, there has not been a lot of work on assuring software whose behaviour is configured using data. There are results on validating the data used in systems that use large quantities of data to perform their function [14, 10] and on safety-related information systems [17]. They describe specification and verification techniques for data, but do not consider systems whose flow of execution is itself determined by data.

Calculation of weakest preconditions is a demanding task; an automated calculator is essential to make it practical and scalable. To our knowledge, there are almost no tools that can handle realistic languages

and their types, except perhaps Java [2]. The extension of a tool like that in [4] to handle a safe language is, therefore, an interesting problem.

Weakest preconditions are the basis of the calculator in [4], which is implemented using the HOL theorem prover [11]. The simple imperative language considered includes recursive procedures; all HOL types are available. A similar calculator is described in [16], but it uses weakest liberal preconditions, which cannot be used to reason about termination.

Termination is also not treated in the more recent approach to invariant calculation in the tool in [15]. Their idea of using patterns of programs to identify invariants, on the other hand, merits further investigation. Given the constrained nature of block-configurable software, it may well be possible to identify a catalogue of program patterns and associated invariants to afford automation.

Availability of tool support is a strong point of the assert-and-verify approach, which is a clear alternative to the technique explored here. For Ada, the SPARK tools merit further investigation to assess automation. For C, the C verifier VCC [5] handles annotated concurrent programs.

Our work is concerned with a component-based verification and assurance technique. Work in this area has typically concentrated on the definition of languages for component connectors, and associated compositional techniques. For example, the approach in [20] advocates verification by model checking in a framework called X-MAN. The work in [19], on the other hand, considers object-oriented models and can be a good basis for description of our work. It is possible that we can specify the block-configurable architecture in the languages for components considered in these works to take advantage of their results for the generation of evidence. We have here, however, dealt directly with (Ada) programs, rather than a high-level modelling language. In addition, these works have not covered the construction of assurance arguments like we do here.

8 Conclusions

We have described how assurance cases can be created for block-configurable software. Our approach maximises resilience of arguments and evidence to expected changes, and enables the build up of reusable assurance-case modules for both components and configuration data. To provide evidence of termination, we have explored the use of weakest preconditions to determine the required constraints. Simple modelling and clearly prescribed adaptations are needed when the set of components changes.

The assurance case demonstrates that the software is "well behaved", by which we mean that the software does not adversely affect the rest of the system. We have identified that the notions of interest for "well behavedness" are related to termination, resources, and exceptions. In this paper we have only considered the argument made for termination. In [13], however, we also explore the arguments for resources and exceptions.

An accurate comparison between the assurance effort entailed by conventional and by block-configurable software is difficult. There is some additional effort required for a block-configurable software. Firstly, the constraints must be determined, but this is offset if a formal approach is used as testing for the particular properties needed in the assurance case is not required. Secondly, the configuration data must be verified. This is expected to be a simple task for the anticipated constraints, and the effort should, therefore, be less than is involved in testing.

For a conventional application, adding new functions requires changes to the code, and the entire program must, therefore, be reverified. For block-configurable software it is required only that the configuration data for the new functions is verified. It is when new functions are integrated into the system that savings in the assurance effort are realised.

A major drawback is the absence of tools to support weakest precondition calculations. This technique can be used to produce evidence for any functional property that can be specified by a postcondition. As we consider more elaborate properties, however, automation becomes more difficult. It is possible to use approximations like in [3]. In this case, we can ensure that invalid data is rejected, but valid data may also be rejected.

Acknowledgements We are grateful to Jane Fenn for her support.

References

1. ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics. International Standard.
2. G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK - A Tool for Validation of Security and Behaviour of Java Applications. In *FMCO*, volume 4709 of *LNCS*, pages 152 – 174. Springer, 2007.
3. A. L. C. Cavalcanti, S. King, C. O'Halloran, and J. C. P. Woodcock. Test-Data Generation for Control Coverage by Proof. *Formal Aspects of Computing*, 2013. Online first. DOI 10.1007/s00165-013-0279-2.
4. A. L. C. Cavalcanti and J. C. P. Woodcock. A Weakest Precondition Semantics for *Circus*. In *Communicating Processing Architectures 2002*. IOS Press, 2002.
5. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOL*, volume 5674 of *LNCS*, pages 23 – 42. Springer, 2009.

6. GSN Standardisation Committee. GSN community standard, November 2011.
7. L. de Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69 – 77, 2011.
8. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
9. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.
10. A. G. Faulkner, P. A. Bennett, R. H. Pierce, I. H. J. Johnston, and N. Storey. The Safety Management of Data-Driven Safety-Related Systems. In *SAFECOMP*, volume 1943 of *LNCS*, pages 86 – 95. Springer, 2000.
11. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
12. Object Management Group. Structured assurance case metamodel (SACM). OMG Standard Document, 2013. OMG Document Number: formal/2013-02-01.
13. R. Hawkins, A. Miyazawa, A. L. C. Cavalcanti, T. Kelly, and J. Rowlands. Assurance Cases for Data-configured Software. Technical report, University of York, Department of Computer Science, York, UK, 2014. Available at `www-users.cs.york.ac.uk/~rhawkins/HMCKR14.pdf`.
14. J. C. Knight, E. A. Strunk, W. S. Greenwell, and K. S. Wasson. Specification and Analysis of Data for Safety-Critical Systems. In *ISSC*, 2004.
15. O. Mraihi, W. Ghardallou, A. Louhichi, L. Labed Jilani, K. Bsaies, and A. Mili. Computing preconditions and postconditions of while loops. In *ICTAC*, volume 6916 of *LNCS*, pages 173 – 193. Springer, 2011.
16. T. Nipkow. Winskel is (almost) Right: Towards a Mechanized Semantics. *Formal Aspects of Computing*, 10(2):171 – 186, 1998.
17. J. Tillotson. System safety and management information systems. In F. Redmill and T. Anderson, editors, *Aspects of Safety Management*, pages 13 – 34. Springer, 2001.
18. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.
19. M. Broy. A core theory of interfaces and architecture and its impact on object orientation. In *International Conference on Architecting Systems with Trustworthy Components*, pages 26–47, 2006. Springer-Verlag.
20. K.-K. Lau and C. M. Tran. X-man: An mde tool for component-based system development. *39th Euromicro Conference on Software Engineering and Advanced Applications*, 0:158–165, 2012.
21. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.