Semi-Automated Design Space Exploration for Formal Modelling[⋆]

Gudmund Grov[1], Andrew Ireland[1], Maria Teresa Llano[2], Peter Kovacs[1]
Simon Colton[2], and Jeremy Gow[2]

[1] Heriot-Watt University, School of Mathematical and Computer Sciences
`{G.Grov,A.Ireland,PK157}@hw.ac.uk`
[2] Goldsmiths College, University of London
`{m.llano,s.colton,j.gow}@gold.ac.uk`

Abstract. Refinement based formal methods allow the modelling of systems through incremental steps via abstraction. Discovering the right levels of abstraction, formulating correct and meaningful invariants, and analysing faulty models are some of the challenges faced when using this technique. We propose *Design Space Exploration* that aims to assist a designer by automatically providing high-level modelling guidance.

Keywords: Design, Abstraction, Event-B, Theory Formation.

1 Introduction

During the development of software intensive systems, the mathematical rigour of *formal methods* brings unique benefits. Specifically, the precision of a formal notation enables design decisions to be clearly communicated and formally verified. However, the use of a formal notation alone is not sufficient to achieve these benefits. Developing design models at the "right" level of abstraction is a creative process, requiring significant skill and experience on the part of the designers. Typically within industrial-scale projects, a design will be modelled at too concrete a level, with the details obscuring the clarity of key design decisions, making it harder to determine if the customer's requirements have been satisfied. In addition, starting with too concrete a design may prematurely "lock" the design team into a particular solution and increase the complexity of the associated formal verification task, i.e. proving properties of the design. Addressing these problems would significantly leverage the creativity of a designer.

We aim at developing a tool that analyses the work of a designer behind the scenes, and automatically suggests design alternatives for Event-B models [1] – alternatives which improve the clarity and correctness of a design. Moreover, a tool that explains for each alternative *what* issue it is addressing and *how* it will effect the design as a whole. The tool will be *semi*-automatic in that while the analysis and synthesis outlined above will be automatic, the designer will remain in full control of the design process. We believe that we can achieve this goal by combining common patterns of modelling with techniques from automated reasoning, in particular automated theory formation. This paper takes the first

[⋆] This work has been supported by EPSRC platform grants EP/J001058/1 and EP/N014758/1, and FP7 WHIM project 611560. We are grateful for feedback on our approach by Jean-Raymond Abrial.

steps towards such tool. As a working example, consider the requirements given below of a simplified protocol for transferring money between bank accounts:

R1: the sum of money across all accounts should remain constant;
R2: transactions can only be completed if the source account has enough funds;
R3: if an amount m is debited from a source account, the target account should be credited by m;
R4: progress should always be possible (no deadlocks).

A designer might choose to represent the protocol as follows in Event-B:

$$\begin{aligned}
&\mathit{start}(a1,a2,m) \overset{def}{=} \textbf{when } a1 \notin \mathit{active}\\
&\qquad \textbf{then } \mathit{pend} := \mathit{pend} \cup \{((a1,a2),m)\} \;||\; \mathit{active} := \mathit{active} \cup \{a1\}\\
&\mathit{debit}(a1,a2,m) \overset{def}{=} \textbf{when } ((a1,a2),m) \in \mathit{pend} \wedge \mathit{bal}(a1) \geq m\\
&\qquad \textbf{then } \mathit{bal}(a1) := \mathit{bal}(a1) - m \;||\; \mathit{pend} := \mathit{pend} \setminus \{((a1,a2),m)\} \;||\\
&\qquad\quad \mathit{trans} := \mathit{trans} \cup \{((a1,a2),m)\}\\
&\mathit{credit}(a1,a2,m) \overset{def}{=} \textbf{when } ((a1,a2),m) \in \mathit{trans}\\
&\qquad \textbf{then } \mathit{bal}(a2) := \mathit{bal}(a2) + m \;||\; \mathit{trans} := \mathit{trans} \setminus \{((a1,a2),m)\} \;||\\
&\qquad\quad \mathit{active} := \mathit{active} \setminus \{a1\}
\end{aligned}$$

The chosen approach involves three steps, each of which is represented through an event that is parametrised by the names of the source ($a1$) and target ($a2$) accounts, along with the value of money (m) associated with the transfer. Step one (event *start*) initiates a transfer by adding the transaction to a *pending* set (*pend*), and uses a set (*active*) to ensure that an account can only be the source of one transfer at a time. Note that || denotes parallel execution. The second step (event *debit*) removes the funds from the source account if sufficient funds exist – *bal* denotes a function that maps an account to its balance. If successful, the transaction is removed from the *pending* set and is added to the *transfer* set. The final step (event *credit*) completes the transaction by adding the funds to the target account, as well as updating the *trans* and *active* sets accordingly. Finally, requirement **R1** is formalised as an invariant, I1: $\Sigma_{a \in dom(bal)} bal(a) = C$ where C is a constant that represents the sum of money across all accounts.

This design abstraction only represents a starting point for the modelling process. A designer will next refine their design ideas through a series of progressively more concrete design abstractions. This gives leverage over the inherent complexity of the design process, enabling the designer to incrementally achieve a customer's requirements. Crucially each refinement step must be formally proved correct. This process is called *correctness-by-construction.* A longer version of this paper is available on ArXiV [5].

2 Towards Design Space Exploration

Key to the style of modelling outlined above is *abstraction* – the ability to create a design at the right level of detail; and to "glue" it to any abstract model through a set of gluing invariants. Trial-and-error is very much part of the expert methodology, where low-level proof failures are examined, and design alternatives in terms of abstractions are experimented with manually (see [2]). Within Design

Space Exploration, our goal is to automate much of the low-level grind associated with the trial-and-error nature of formal modelling, and provide a designer with *high-level* modelling advice in real-time.

In particular, we aim to generate alternative models at a higher level of **abstraction** than the original model to deal with a flaw. The intuition is that the flaw is a result of being too concrete. Moreover, within a correct abstraction, the designer has the additional burden of correctly defining the system behaviour and supplying numerous auxiliary invariants that are required for the formal verification process. To support this, we will suggest **adaptations** of the initial model at the same level of abstraction. This could be for instance in terms of additional invariants, or even changes to the behaviour of the system. As can be seen in the next section, unconstrained generation of new models will result in an enormous search space which will be infeasible in practice. Instead, the approach we are proposing has two phases, **analysis** and **generation**, which will iterate until a satisfactory solution is found, possibly including user input.

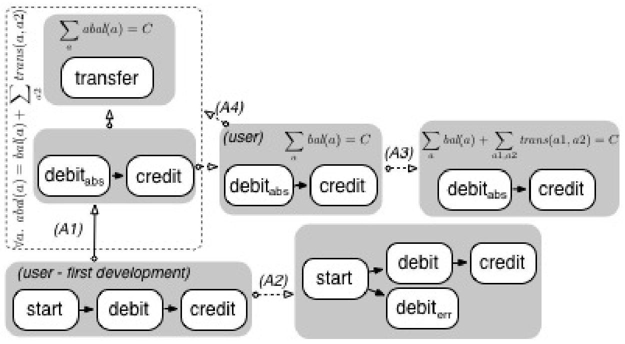
Analysis phase Automated Theory Formation (ATF) is a technique that invents concepts to describe and categorise examples from the input domain, makes conjectures which relate the concepts, and seeks proofs and counterexamples to determine the truth of the conjectures. The HR ATF system [3] will be used in the **analysis** phase to explore given Event-B models and highlight problematic areas. A major challenge will be to find heuristic techniques that effectively prune the design space so that a designer is presented with a useful set of modelling alternatives. This analysis will aim to pin-point both *where* and *what* the problem may be in order to guide the generation phase, and to identify the most interesting solutions. Our approach will be a significant evolution of our previous work on using HR for Event-B [4, 9], where we will explore unrestricted theories and include event information in order to explore hypotheses related to the events. We explore simulation traces derived from simulating models, to identify conjectures that are associated with failed steps from the simulation trace. This strategy has proven successful as evidenced in [4], and is extended here by including event information. This will indicate that a variable or an event are associated with failures in the model and therefore should be the focus for the generation phase, as will be illustrated in §3. This section also illustrates how HR can be used to exploit erroneous user given invariants in order to suggest adaptations of them. We will also search for invariants that are required in order to prove the consistency between the abstract and concrete models; i.e. gluing invariants, which we have already explored in [9]. Finally, we will exploit HR's support for the generation of near conjectures, i.e. conjectures that are true for a percentage threshold of the examples they have. Building upon this functionality, we will explore how this can be tailored to the needs of formal modelling. That is, although formal methods are typically based on definite answers, e.g. a property is either true or false, we believe that a weaker notion of truth is called for when exploring design alternatives, what we call *near-properties*; i.e. properties that are true for most, but not all, behaviours, e.g. *"event X always violates invariant I, but it is always re-established by event Y"*. Paying attention to such properties can lead to insights and in particular suggest solutions which lie just beyond the fringe of what is currently *true* about a design.

Generation phase The results of the analysis phase are then used in the model **generation** phase, where alternative abstractions and adaptations of the model are generated. The system must be able to 'explore' design alternatives also for new and previously unseen scenarios. The component that performs the actual generation of new abstractions and adaptations can therefore not be too prescriptive, as was the case with our reasoned modelling critics [6]. For his (unpublished) honours dissertation, one of the authors (Kovacs) has made the first step towards such a component by implementing a generic framework for model generation as a plug-in to the Rodin tool-set [7]. The key feature of this plug-in is that it has a layered design: at the bottom is a set of low-level but generic 'atomic operators' that make small changes to a model, e.g. 'delete variable' and 'merge events'. These atomic operators can then be combined in order to generate new models, and constrained to reduce the number of possible models generated. It is up to the system to find the right combination of operators and to constrain them in the best possible manner. Thus, a "complete" set of atomic operators would allow the generation of all possible alternative models. This gives flexibility to our proposed approach to Design Space Exploration, enabling us to handle new and unforeseen circumstances. Due to space constraints, the details of this tool has been omitted and we refer the interested reader to [5, 7]. In §3 we give examples of how this framework is used.

Common patterns of modelling As will be illustrated in §3, common modelling patterns will play a central role in finding the right combination of operators. These will be at a very high-level to enable flexibility in terms of their application and therefore enable us to provide assistance in situations where there are no applicable design patterns. The analysis will be used to suggest suitable patterns and guidance as to how they can be implemented. To support this, we have already identified several *refinement patterns* [4] in previous work; however as we cannot refine away flaws, this will be applied in inverse, essentially turning them into *abstraction patterns.* Some abstraction patterns have also been identified and represented using the operator framework in [7]. The experiments in the next section are utilising two patterns: (1) "undoing" bad behaviour by introducing a special **error** (or exception) **case**; and (2) **abstracting away** the problem when it can be pinpointed between certain events. This amounts to "atomising" sequential events into a single event.

3 Illustrative examples and initial experiments

In terms of realising our vision we have undertaken experiments at the level of analysing design models as well as mechanising generation. We present these experiments next. The selection of operators and the integration of the two phases is currently manual; our ultimate goal is to automate the full development chain.



Starting from the initial development, abstraction (A1) and adaptation (A2) are suggested to deal with violation of requirement R4. Given that I1 is a near-invariant, a new invariant is suggested in (A3); or an abstraction (A4) with the required gluing invariant.

Fig. 1. A diagrammatic summary of a small design space exploration.

Consider again the user provided model of a money transfer protocol given in §1. As it stands, the model is flawed since **R4** is violated when all accounts have started a transaction but none of the source accounts have sufficient funds. Moreover, event *debit* violates invariant I1 since the amount removed from the source account is not accounted for in the invariant, which breaks requirement **R2**. Our aim in such situations will be to offer the designer modelling alternatives that address the flaws. Figure 1 summarises the alternatives generated through our approach, and below we outline how this was achieved. More details can be found in the long version of the paper [5].

Abstraction A1 The first step of the analysis is to generate simulation traces by running the ProB simulator [8], which will also check if the invariants hold. This is input for HR which will use the concept *good* for states in which ProB did not find any invariant violations. HR is then used to search for properties that involve the concept ¬*good*, and this analysis suggest that the generation of bad states are associated to event *debit* and variable *active*.

We can apply the "abstract away" pattern to this violation. One implementation of this pattern is to remove the variable that two (sequential) events use to communicate an intermediate result, and then combine this sequence into an atomic event. A naive application of this pattern in our operator framework will generate 12 alternatives, however by constraining the generation to always include the event *debit* and variable *active*, this is reduced to 2 alternatives (thus pruning the search space by 83%), one of them being the desired abstraction:

$$\begin{array}{ll} debit_{abs}(a1, a2, m) \overset{def}{=} & \textbf{when } a1 \notin active \land bal(a1) \geq m \\ & \textbf{then } active := active \cup \{a1\} \mid\mid bal(a1) := bal(a1) - m \\ & \quad\quad \mid\mid trans := trans \cup \{((a1, a2), m)\} \end{array}$$

Adaptation A2 An alternative analysis is to apply the error-case pattern. Intuitively, this means introducing a new "error-handling" event that will "undo" some previous state changes when the desired path is not applicable. This can be implemented so that it reverses a previous action in cases when an event of the desired path stays disabled. This require transformations to negate an event's guard, reverse an action of an event and combine the guards of one event with the actions of another. Here a naive implementation will generate 10 alternatives, while if we apply the same constraints as in (A1) then this is reduced to 7, including the generation of the error-handling event:

$$\begin{aligned} \mathit{debit}_{err}(a1, a2, m) \stackrel{def}{=} \ & \textbf{when } ((a1, a2), m) \in \mathit{pend} \land \mathit{bal}(a1) < m \\ & \textbf{then } \mathit{pend} := \mathit{pend} \setminus \{((a1, a2), m)\} \ || \\ & \qquad \mathit{active} := \mathit{active} \setminus \{a1\} \end{aligned}$$

debit_{err} handles the case when the source account does not have enough funds.

Adaptation A3 Let's assume the user selects **A1**. Through analysis of this alternative, invariant (I1) is still violated and HR is re-applied. Through manual inspection of the result of HR, we can see that we are in a "bad state" when *trans* and *active* are not empty, i.e. when there are transactions currently in progress. As a results HR is re-applied to search for conjectures that involve the concepts *trans* and *active* as well as the invariant itself; i.e. $C = \Sigma_{a \in dom(bal)}\ bal(a)$. HR is then able to generate an *adaptation* of the invariant I1 that addresses the violation by debit_{abs}. Note that this adaptation is achieved by including the "internal state" *trans* within the invariant. The Event-B representation of the invariant, which replaces I1, is:

$$\text{I2: } \Sigma_{a \in dom(bal)} bal(a) + \Sigma_{(a1,a2) \in dom(trans)} trans(a1, a2) = C$$

Abstraction A4 Although correct, invariant I2 is not a natural representation of **R1**, as compared with near-invariant I1. The designer may wish to explore an alternative abstraction in which I1 is an invariant. Our final alternative **A4** represents such an abstraction. Based on the output given by HR for alternative **A1**, we can re-apply our "abstract away" pattern, albeit with a slighly modified implementation that deletes two variables. Unconstrained, this operator will generate 6 possible alternatives, while a constrained application, which takes into account the analysis, only generates 2 alternatives, one of them being the desired *transfer* event[3]:

$$\begin{aligned} \mathit{transfer}(a1, a2, m) \stackrel{def}{=} \ & \textbf{when } \mathit{abal}(a1) \geq m \land a1 \neq a2 \\ & \textbf{then } \mathit{abal}(a1) := \mathit{abal}(a1) - m \ || \\ & \qquad \mathit{abal}(a2) := \mathit{abal}(a2) + m \end{aligned}$$

Finally, in order to prove the consistency between the abstract and concrete models, a gluing invariant is required. Therefore, we enter again in an analysis phase where HR is used to form a theory of the refinement step and search for the invariant. HR is able to figure out the relation between the abstract variable

[3] Technically, the Event-B syntax of the action should be: $\mathit{abal} := \mathit{abal} \mathbin{\lhd\!\!-} \{a1 \mapsto \mathit{abal}(a1) - m, a2 \mapsto \mathit{abal}(a2) + m\}$

abal and the concrete representation; i.e. variables *bal* and *trans*. Part of our future work will be focused on tailoring HR for the formal methods context so that invariants such as the gluing invariant required in this refinement step can be formed.

4 Conclusion and future work

Focusing on Event-B, we have introduced our approach to *Design Space Exploration* for formal modelling, supported by an initial implementation with partly automated experiments. Currently, the sub-components of our approach are partly automated, while their integration is manual. HR has to be manually guided and we have to manually inspect its output as well as select and combine the relevant operators to perform the generations. Our goal is to fully automate all parts, and provide users with a list of new (and ideally ordered by perceived relevance) modelling alternatives. The approach is *semi*-automatic in that the user will decide on how to use the alternatives. In this paper we have provided the first step towards realizing our goal and have shown the feasibility of the overall approach. However, there is still a long way to go: we have already discussed the desirable features for the analysis phase; in addition, we need to identify a sufficiently small, yet complete, set of atomic operators, constraints and combinators, in order to be able to generate all necessary alternatives in the generation phase. It is crucial that these are controlled to avoid generating duplicates. The phases must then be integrated to be able to automate the selection and combination of operators based upon the analysis. The level of support we aim to provide is very ambitious. If successful, our approach will increase the productivity and accessibility of Event-B, but more importantly, it will provide valuable insights into how formal methods can be deployed more widely.

References

1. J.-R. Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.
2. M. Butler and D. Yadav. An incremental development of the mondex system in Event-B. *Formal Aspects of Computing*, 20(1):61–77, 2008.
3. S. Colton. *Automated Theory Formation in Pure Mathematics.* Springer, 2002.
4. G. Grov, A. Ireland, and M. T. Llano. Refinement plans for informed formal design. In *ABZ*, volume 7316 of *LNCS*, pages 208–222. Springer, 2012.
5. G. Grov, A. Ireland, M. T. Llano, P. Kovacs, S. Colton, and J. Gow. Semi-Automated Design Space Exploration for Formal Modelling. arXiv:1603.00636.
6. A. Ireland, G. Grov, M. Llano, and M. Butler. Reasoned modelling critics: turning failed proofs into modelling guidance. *SCP*, 78(3), 2013.
7. P. Kovacs. Automating abstractions in formal modelling, 2015. Heriot-Watt University, Undergraduate Honors Thesis. Available from `http://bit.ly/1JnL0Ts`.
8. M. Leuschel and M. J. Butler. ProB: A model checker for B. In *Proceedings of Formal Methods Europe 2003*, pages 855–874, 2003.
9. M. T. Llano, A. Ireland, and A. Pease. Discovery of invariants through automated theory formation. *Formal Aspects of Computing*, 2012.