Ensuring Extensibility within Code Generation

C. J. Lovell, A. Edmunds, R. Silva, I. Maamria and M. Butler

School of Electronics and Computer Science, University of Southampton, SO17 1BJ, UK
cjl3@ecs.soton.ac.uk

Making the step from Event-B to code is a process that can be aided through automatic code generation. The code generation plug-in for Rodin is a new tool for translating Event-B models to concurrent programmes. However users of such a tool will likely require a diverse range of target languages and target platforms, for which we do not currently provide translations. Some of these languages may be subtly different to existing languages and only have modest differences between the translation rules, for example C and C++, whilst others may have more fundamental differences. As the translation from Event-B to executable code is non-trivial and to reduce the likelihood of error, we want to generalise as much of the translation as possible so that existing translation rules are re-used. Therefore significant effort is needed to ensure that such a translation tool is extensible to allow additional languages to be included with relative ease. Here we concentrate on translation from a previously defined intermediary language, called IL1, which Event-B translates to directly.

The intermediary language IL1 is an EMF metamodel representation of generic properties and functionality found in many programming languages. It has representations for key structural concepts such as variables, subroutines, function calls and parameters. The translation of predicates and expressions contained within the code are handled by a new extension to the theory plug-in, which allows translation rules to be developed for specific target languages within the Rodin environment. The generic nature of the intermediary language is designed to allow for a wide range of different target languages. Developers of new target languages are required to write translators in Java for the conversion from the EMF representation to the code of their target language. To do this we provide a central translation manager, that takes an IL1 model and automatically calls the appropriate translators for each element of the model, whilst also providing the link to the predicate and expression translators provided by the new theory plug-in. The developer registers their translators for the target language through an extension point, where currently there are 15 light-weight translators required for a new target language. To aid the developer, we provide abstract translators for each required element in the IL1 model that has to be translated. These translators perform the majority of the translation automatically, meaning that in most cases all the developer is required to do is format strings into the appropriate structure for their target language. For example in an branch statement, the developer would be required to write a method stating how a branch is defined and structured in their language, using a set of previously translated guard conditions and actions. Importantly, the flexibility remains for the developer to re-write any of the translations if the ones provided are not suitable. To test our approach, we have built translators for C, Ada and Java using the same underlying abstract translators.

Additionally we consider the case where a new language may be required that has only modest differences to an existing language. A good example of this is to consider the case where a different library may want to be used from one used in an existing translation. For instance in C, concurrency can be achieved through different mechanisms such as OpenMP or Pthreads. In this case it may be that all but the mechanism for handling a subroutine call are the same, meaning that the majority of the translation can occur using common translators, with separate translators for the different methods of handling a subroutine call. To allow for this we allow the developer to assign a core and specialisation language to each translator they build. In cases where a translator for the specialisation language does not exist, the translator will automatically defer to the default core language translator, if one exists. This means that default translators for a particular core langauge can be written for the majority of the translation, with specialisations being provided where differences occur. The core and specialisation of the language is also reflected in the theory translator, meaning that language theories are only required for the core languages, rather than for each individual specialisation.

Part of this research was carried out within the European Commission ICT DEPLOY contract / grant number: 214158.