Refinement as Inclusion of Predicates over Programs

Alessandro Coglio

Kestrel Institute
http://www.kestrel.edu/~coglio

July 2012

Abstract

An approach to refinement is described: (1) formalize the syntax and semantics of the target programming language; (2) specify the requirements by defining a predicate over programs that characterizes the possible implementations; (3) refine the specification stepwise by defining monotonically decreasing predicates over programs, according to decisions that narrow down the possible implementations; (4) conclude with a predicate that characterizes a unique program in explicit syntactic form, from which the program text is obtained. These activities are carried out using a general-purpose logical language, preferably inside a theorem prover, ideally extended with automated transformations and with a very simple code generator. The approach is more direct than existing refinement formalisms.

1 Refinement in the Literature

1.1 Origins

The idea of deriving an implementation from a specification via a sequence of transformations was alluded to in [14], proposed in [11], and called 'refinement' in [18]. The related idea of using theorem proving to generate an implementation, along with its correctness proof, from a specification was pioneered in [12] and [17].

1.2 Refinement Formalisms

Several refinement formalisms exist, e.g. [9]. A refinement formalism includes:

1. a set S of specifications, used to describe requirements and also intermediate derivation stages towards implementation;
2. an 'is refined by' (or 'is transformed into') preorder $\rightsquigarrow$ over S, used to link derivation stages.

A preorder is a reflexive and transitive relation: reflexivity ($s \rightsquigarrow s$) means that each specification is a (trivial) refinement of itself; transitivity ($s \rightsquigarrow s' \wedge s' \rightsquigarrow s'' \Rightarrow s \rightsquigarrow s''$) means that refinements can be chained.

In order to obtain implementations, there should be also a set $S_{\mathrm{E}} \subseteq S$ of "executable" specifications, along with a code generation function $\mathcal{C} : S_{\mathrm{E}} \rightarrow P$, where P is a set of programs in some programming language like C or Java. $\mathcal{C}$ may be part of a tool based on the refinement formalism, rather than part of the refinement formalism per se. There may be several functions $\mathcal{C} : S_{\mathrm{E}} \rightarrow P$, $\mathcal{C}' : S'_{\mathrm{E}} \rightarrow P'$, $\mathcal{C}'' : S''_{\mathrm{E}} \rightarrow P''$, ... (e.g. for different programming languages), but one suffices for the present discussion. This general framework includes the special case of an executable specification language ($S_{\mathrm{E}} = S$) and the special case of a specification language that extends a programming language ($S \supseteq P$ and $\mathcal{C}$ is a no-op).

1.3 Derivations

A derivation by refinement is a sequence

$$s_0 \rightsquigarrow s_1 \rightsquigarrow \cdots \rightsquigarrow s_n \overset{\mathcal{C}}{\mapsto} p \tag{1}$$

where s_0 captures requirements, $s_1, \ldots, s_n$ are intermediate stages, and the implementation p is generated via $\mathcal{C}$ from $s_n \in S_{\mathrm{E}}$. Since $\rightsquigarrow$ is transitive, $s_0 \rightsquigarrow s_n$.

A function $\mathcal{I} : S \rightarrow 2^P$ associates a specification with the set of implementations that can be derived from the specification by refinement:

$$\mathcal{I}(s) = \{\mathcal{C}(s') \in P \mid s \rightsquigarrow s' \wedge s' \in S_{\mathrm{E}}\}. \tag{2}$$

Since $\rightsquigarrow$ is transitive, $\mathcal{I}$ is monotone:

$$s_1 \rightsquigarrow s_2 \;\Rightarrow\; \mathcal{I}(s_1) \supseteq \mathcal{I}(s_2). \tag{3}$$

PROOF: Let $p \in \mathcal{I}(s_2)$. From (2), there exists $s' \in S_{\mathrm{E}}$ such that $p = \mathcal{C}(s')$ and $s_2 \rightsquigarrow s'$. From the hypothesis $s_1 \rightsquigarrow s_2$ and the transitivity of $\rightsquigarrow$, $s_1 \rightsquigarrow s'$. From (2), $p \in \mathcal{I}(s_1)$. Since p is arbitrary, $\mathcal{I}(s_1) \supseteq \mathcal{I}(s_2)$. QED

Since $\rightsquigarrow$ is reflexive, the possible implementations of an executable specification include the one generated from the specification:

$$s \in S_{\mathrm{E}} \;\Rightarrow\; \mathcal{C}(s) \in \mathcal{I}(s). \tag{4}$$

PROOF: From the reflexivity of $\rightsquigarrow$, $s \rightsquigarrow s$. From the hypothesis $s \in S_{\mathrm{E}}$ and (2), $\mathcal{C}(s) \in \mathcal{I}(s)$. QED

Because of (3) and (4), the derivation (1) is mirrored by a sequence of monotonically decreasing sets of programs

$$\mathcal{I}(s_0) \supseteq \mathcal{I}(s_1) \supseteq \cdots \supseteq \mathcal{I}(s_n) \supseteq \{p\}. \tag{5}$$

This suggests the view that a derivation by refinement is a sequence

$$\widehat{p}_0 \supseteq \widehat{p}_1 \supseteq \cdots \supseteq \widehat{p}_n \supseteq \{p\} \tag{6}$$

where $\widehat{p}_0 \in 2^P$ are the possible implementations of the requirements, each $\widehat{p}_i \subseteq \widehat{p}_{i-1}$ narrows down the possible implementations, and $p \in \widehat{p}_n \subseteq \widehat{p}_0$ is the chosen implementation.

1.4 Indirection

In this view, a refinement formalism is merely an instrument to construct, indirectly, a derivation (6), constrained to have the form (5), where $s_0, s_1, \ldots, s_n$ and p are linked by (1). This indirection implies that the use of any specific refinement formalism may limit, in particular, the choice of $\widehat{p}_0$, which must have the form $\mathcal{I}(s_0)$ for some s_0: thus, $\widehat{p}_0$ may include undesired implementations or exclude desired implementations.

2 A More Direct Approach

The indirection disappears by taking:

- $S = 2^P$, i.e. a specification is a set of programs;
- $\rightsquigarrow \; = \; \supseteq$, i.e. refinement is set inclusion;
- $S_{\mathrm{E}} = \{\{p\} \mid p \in P\}$, i.e. an executable specification is a singleton set;
- $\mathcal{C}(\{p\}) = p$, i.e. code generation is essentially a no-op.

Under these conditions, $\mathcal{I}(\widehat{p}) = \widehat{p}$. PROOF: $\mathcal{I}(\widehat{p}) = \{\mathcal{C}(\widehat{p}') \in P \mid \widehat{p} \rightsquigarrow \widehat{p}' \wedge \widehat{p}' \in S_{\mathrm{E}}\} = \{\mathcal{C}(\{p\}) \in P \mid \widehat{p} \supseteq \{p\} \wedge p \in P\} = \{p \in P \mid p \in \widehat{p}\} = \widehat{p}$. QED

However, literally using 2^P (or anything isomorphic to it) as S is infeasible: if, as customary, P is countably infinite, then 2^P is uncountable. But it is feasible to use, as specifications in S, predicates defined over programs and written in a general-purpose logical language $\mathcal{L}$ like higher-order logic [10]. A predicate is a finite term that denotes a set. Predicates are only countable, but sets of programs that are not denoted by any predicate are unlikely to be of practical interest. Refinement is predicate inclusion, expressed via logical implication. The set S_{E} of executable specifications should be restricted to predicates that not only denote singleton sets, but also have a form from which the program text can be obtained via a simple $\mathcal{C}$.

The approach consists of the following activities:

1. Formalize the syntax and semantics of the target programming language.
2. Specify the requirements by defining a predicate over programs that characterizes the possible implementations.
3. Refine the specification stepwise by defining monotonically decreasing predicates over programs, according to decisions that narrow down the possible implementations.
4. Conclude the derivation with a predicate that characterizes a unique program in explicit syntactic form, from which the program text is obtained.

These activities occur within $\mathcal{L}$. In particular, the correctness of each refinement step is proved in the logic of $\mathcal{L}$.

These activities are best carried out inside a general-purpose theorem prover, i.e. a software tool that provides a general-purpose logical language (i.e. $\mathcal{L}$) and inference mechanisms for the logic of the language. Prime exemplars are higher-order logic theorem provers like Isabelle/HOL [3] and Coq [2]. Sections 2.3.6 and 2.4.2 describe theorem prover extensions that are useful for refinement.

The following subsections elaborate the four activities listed above. It is assumed that $\mathcal{L}$ is higher-order logic, includes a type Bool for boolean values, and supports user-defined types and constants; a function from a type T to a type T' is a constant of function type $\mathsf{T} \to \mathsf{T}'$; a predicate over a type T is a function of type $\mathsf{T} \to \mathsf{Bool}$. It is also assumed that the four activities are carried out inside a theorem prover.

2.1 Formalize the Target Programming Language

2.1.1 Syntax

The documentation of a programming language usually includes a grammar that describes the concrete syntax. Compilers and other language processing tools generally use a more convenient abstract syntax internally. For similar convenience, an abstract syntax of the target programming language should be formalized. This typically consists of inductively defined types that capture the constructs of the language—expressions, statements, declarations, compilation units, etc. A type Prog_A captures the set P of target programs, viewed as abstract syntactic trees.

Optionally, the concrete syntax of the target programming language can be formalized, along with the grammar and with predicates that associate program texts (i.e. sequences of characters) with their corresponding parse trees and abstract syntax trees. A type Prog_C captures the set P of target programs, viewed as texts. A predicate parse of type $\mathsf{Prog}_\mathrm{C} \rightarrow \mathsf{Prog}_\mathrm{A} \rightarrow \mathsf{Bool}$ associates program texts with their corresponding abstract syntax trees.

Some specialized programming languages are not (entirely) textual but contain graphical elements, e.g. Stateflow [7].[1] It is possible to formalize an abstract syntax that includes abstract representations of graphical elements (e.g. boxes and arrows). It is also possible to formalize a concrete syntax that includes physical layout information of graphical elements (e.g. sizes and positions). If the language has a textual import/export format, it can be formalized in the same way as entirely textual languages.

The documentation of a programming language usually also includes syntactic constraints that must be satisfied as a pre-requisite for execution and that are generally checked by compilers or interpreters. Examples are uniqueness of identifiers (in each name space) and type correctness. These constraints are sometimes called 'static semantics', while 'dynamic semantics' describes execution. These constraints must be formalized, typically as predicates over the types that formalize the abstract syntax, including a predicate check of type $\mathsf{Prog}_\mathrm{A} \rightarrow \mathsf{Bool}$.

2.1.2 Semantics

The semantics of the target programming language is typically formalized by defining types that capture (components of) execution states (values, objects, variables, call stacks, concurrent threads, etc.), along with functions that describe how the constructs of the abstract syntax operate on these execution states. Denotational semantics and operational semantics are well-known styles.

Some programming languages include constructs whose semantics is platform-dependent, e.g. the size of integer types in C. The formalized semantics should be parameterized over such platform-dependent aspects. Requirements on target programs can be expressed either over all possible instantiations of the parameters, leading to universally portable programs, or over some instantiations only, leading to programs that are guaranteed to meet the requirements only on the platforms described by these instantiations.

2.1.3 Subset

Not every program uses every feature of its language. Thus, formalizing a suitably large subset of the language may suffice to target all the programs

[1]Languages like Stateflow are often considered more specification/modeling languages than programming languages, because they are higher-level and they are compiled to conventional programming languages like C. Nonetheless, deriving a Stateflow model/program via refinement from higher-level requirements is conceivable.

of interest. If the derivation of a new program needs a feature that is not formalized, the subset can be extended.

The subset can exclude, or restrict in scope, features whose platform dependencies are complex to formalize, e.g. raw memory access in C. Along similar lines, the language formalization may “hard-wire” some platform-dependent features to specific platforms instead of being parameterized over them, trading generality for simplicity.

2.1.4 Libraries

Mundane programming languages come with (often extensive) libraries, whose semantics must be formalized in order to target programs that use such libraries. Since not every program uses every part of every library, it suffices to formalize the subset of libraries that is used, extending the subset as needed for new program derivations.

Some libraries interact with external devices like keyboard and screen. Formalizing the semantics of such libraries involves capturing these interactions at an appropriate level of abstraction, which depends on the requirements to be specified for the interactions between target programs and external devices via these libraries. For example, the formalized execution states could include traces of input/output events (e.g. keystrokes, whole lines of text read from or written to a terminal, GUI menu selections) and the formalized semantics of the libraries could describe how they manipulate such event traces.

Some libraries involve non-functional aspects like time, e.g. a Java method that waits for a given number of milliseconds. These aspects are discussed next.

2.1.5 Non-Functional Aspects

In order to specify non-functional requirements that involve memory, time, power, etc. of target programs, the formalized semantics of the programming language and of its libraries must include suitable notions. For example, functions could be defined that associate the formalized execution states with their size in memory. As another example, functions could be defined that associate the formalized language constructs and library calls with their execution times and with their energy usage.

These non-functional aspects often have complex platform dependencies. The formalization can parameterize these non-functional aspects over suitable platform attributes, e.g. the size of each elementary data type, from which the size of larger data types and of execution states can be calculated. The formalization can also relax exact values to ranges or probability distributions, e.g. for the execution time and energy usage of each language construct, enabling worst-case or probabilistic requirements to be specified.

2.1.6 Lower-Level and Hardware Languages

A derivation by refinement typically targets relatively high-level programming languages like C or Java. But it can also target lower-level languages like JVM bytecode [13], LLVM intermediate representation [5], and assembly. Formalizing the syntax and semantics of these lower-level languages is often easier than formalizing the syntax and semantics of higher-level languages.

Going further down the computing stack, a derivation by refinement can target hardware "programs" written in hardware "programming" languages like VHDL [8]. The line between software and hardware is blurred. Hardware languages can be formalized similarly to software languages.

2.1.7 Multiple Languages

Some programs consist of portions written in different programming languages, communicating via inter-language mechanisms like JNI [4] and CORBA [1]. In order to target such multi-lingual programs, it is necessary not only to formalize the individual languages, but also their interaction.

Syntax is easily handled. For example, given types $\mathsf{JProg}_{\mathrm{A}}$ and $\mathsf{CProg}_{\mathrm{A}}$ for (abstract syntax trees of) Java and C programs, a product type $\mathsf{JCProg}_{\mathrm{A}} = \mathsf{JProg}_{\mathrm{A}} \times \mathsf{CProg}_{\mathrm{A}}$ captures Java/C programs. If concrete syntax is formalized, there is also a product type $\mathsf{JCProg}_{\mathrm{C}} = \mathsf{JProg}_{\mathrm{C}} \times \mathsf{CProg}_{\mathrm{C}}$. This generalizes to three or more languages.

The semantics of multi-lingual programs involves the semantics of the inter-language mechanisms, e.g. data marshalling and unmarshalling. Since not every multi-lingual program uses every feature of the (sometimes rich) inter-language mechanisms, it suffices to formalize the subset of features that is used, extending the subset as needed for new program derivations.

2.1.8 Validation

Since programming languages are documented mostly informally (except for the use of grammars and for unusual cases like Standard ML [15]), generally the formalization of a language cannot be formally proved correct. Nevertheless, there are ways to validate the formalization, i.e. increase confidence that it adequately captures the language:

- Follow the documentation as closely as possible, using the same terminology in the formalization and explicitly tracing, via comments, each part of the formalization to well-defined parts of the documentation. This reduces errors at the outset and facilitates subsequent review.
- If the documentation includes formal parts, convert them into $\mathcal{L}$ and either use them as part of the formalization or prove that they follow from the

formalization. This process may involve formalizing in $\mathcal{L}$ the formalisms in which the formal documentation is expressed (see Section 2.2.5).

- State and prove expected theorems about the language, e.g. type safety. For instance, a theorem could say that if check (see Section 2.1.1) holds, then during execution the value assigned to each variable has the variable's type.
- Convert test suites into logical formulas and prove them. The theorem prover "executes" the tests, at the formalization level instead of the implementation level.

2.1.9 Reuse

The formalizations of programming languages, libraries, and inter-language mechanisms can be stored into $\mathcal{L}$ libraries and used in multiple derivations by refinement. The same formalizations can also be used for other purposes, e.g. semantic clarification and program verification.

2.2 Specify the Requirements

2.2.1 Programs

The requirements are specified by defining a predicate spec_0 of type $\mathsf{Prog} \to \mathsf{Bool}$. The choice of Prog is also part of the requirements: the target programs can be in C, Java, or any other language (formalized as explained in Section 2.1), can be single- or multi-lingual (see Section 2.1.7), and can consist of abstract or concrete syntax (see types Prog_A and Prog_C in Section 2.1.1). In order to allow implementations in any one of a set of languages, Prog can be defined as the disjoint union of the types of programs in such languages, e.g. $\mathsf{Prog} = \mathsf{JProg} + \mathsf{CProg}$ if implementations in Java and C are equally acceptable.

If the programs in Prog consist of abstract syntax, spec_0 should include the satisfaction of check (see Section 2.1.1), ensuring that each possible implementation of the requirements can be at least executed. If the programs in Prog consist of concrete syntax, spec_0 should include the existence of an abstract syntax tree that is related to the program text via parse (see Section 2.1.1) and that satisfies check, ensuring that each possible implementation of the requirements can be at least parsed and executed.

spec_0 may include structural constraints that define required syntactic interfaces with external code. For example, a Java program that implements a cryptographic library may be required to consist of a single package with a certain name and with certain public classes (e.g. for keys) and public methods (e.g. to encrypt and decrypt). As another example, a C++ program that implements a GUI wrapper of a physical simulator written in C, may be required to call certain simulator top-level functions—which are not part of the target C++ program.

2.2.2 Target-Independent Part

spec_0 typically includes requirements on the execution of the program. These are best expressed by incrementally defining auxiliary types and constants (which include functions and predicates) that culminate with spec_0. Some of these auxiliary types and constants may be independent from the choice of the target program and language.

For example, the specification of the Java cryptographic library mentioned in Section 2.2.1 may include types Key and Block for keys and data blocks, and functions $\mathsf{encrypt}$ and $\mathsf{decrypt}$ of type $\mathsf{Key} \rightarrow \mathsf{Block} \rightarrow \mathsf{Block}$. These are $\mathcal{L}$ types and functions, not Java classes and methods. They are used to impose requirements on the target Java classes and methods (see Section 2.2.3) but are independent from them. They can be used equally well to impose requirements on C types and functions, if C is the target language.

As another example, the specification of an operating system may include an abstract model of states and system calls that is used to impose requirements on a C implementation of the operating system as well as on a Java simulator of the operating system.

In general, the target-independent part of a specification can be reused to define different predicates spec_0, spec'_0, spec''_0, ... over program types Prog, Prog', Prog'', ...

2.2.3 Target-Dependent Part

The target-dependent part of a specification consists of types and constants, including spec_0, that "connect" the target-independent part to entities of the target programs.

For instance, the connection between the functions $\mathsf{encrypt}$ and $\mathsf{decrypt}$ mentioned in Section 2.2.2 and the corresponding Java methods is that the functions describe the relationships between the input data and the output data of the methods. In expressing this connection, the target-dependent part of the specification can require the methods to either return newly created objects that contain the output data, or overwrite the input data with the output data in the objects passed as arguments. Both styles are consistent with the pure mathematical functions $\mathsf{encrypt}$ and $\mathsf{decrypt}$ (which contain no information about Java object creation or update), but they make a difference to code that uses the cryptographic library. Thus, the choice of the style may be an important requirement. Less strict requirements that allow either style can be expressed using disjunctive constraints.

If the target programs are required to use external code that is not part of the standard libraries of the target programming language, the target-dependent part of the specification must include some model of that code and some connection between that code and the target programs. For instance, consider the C++ GUI mentioned in Section 2.2.1. Some details of the physical simulator may be irrelevant to the GUI. The target-independent part of the specification should include a suitably abstract model of the simulator and a description of the GUI in terms of this abstract model, e.g. to draw a graph of a certain quantity over time—regardless of the (simulated) physical laws that control the quantity. The target-dependent part of the specification should include a model of the top-level C functions of the physical simulator in terms of its abstract model, e.g. a function may advance the simulation to the next time instant and another function may retrieve the current value of a quantity. This creates the necessary connection, at the requirement level, between the GUI wrapper (the target program) and the simulator (the external code).

The target-dependent part of a specification can be informally viewed as a "refinement" of the target-independent part, in the sense that it adds detail that can be regarded as implementation decisions. For example, the Java methods for encryption and decryption discussed above implement the functions encrypt and decrypt, in some sense. However, in the approach to refinement described in this paper, the requirement specification is complete, and refinement starts, only when spec_0 is completely defined, which includes the target-dependent part. This makes all the requirements explicit in the specification, as opposed to folding some of them into the refinements.

2.2.4 Non-Functional Requirements

The fact that spec_0 is a predicate over programs enables the expression of certain non-functional requirements, e.g. on code size, use or non-use of certain constructs or libraries or interfaces, cyclomatic complexity, call stack depth, computational complexity, parameter/result passing style, and even indentation (if the concrete syntax of programs is formalized). These requirements are typically hard to capture in existing refinement formalisms, which constrain target programs indirectly, as explained in Section 1.4: if the specification does not capture these requirements, a derivation may still produce an implementation that satisfies them, but in order to assess that, it is necessary to examine not only the specification, but also the derivation or implementation.

Other non-functional requirements, which involve memory, time, power, etc., can be included in the target-independent part of the specification. For example, the abstract model of the operating system mentioned in Section 2.2.2 could include a function that associates a maximum time duration to each system call. The target-dependent part of the specification of the C implementation of the operating system could require each C function that realizes a system call to terminate execution within the maximum time duration. To express these non-functional requirements, the semantics of the target programming language must include suitable notions (see Section 2.1.5).

Constraints on the use of external libraries and interfaces can prevent covert activities that are notoriously hard to exclude at the specification level. For instance, requiring a password management application to not use any network library from the underlying operating system, ensures that the application does not leak passwords to remote sites through those libraries.

2.2.5 Specialized Formalisms

Certain requirements are conveniently expressed using specialized formalisms, e.g. finite state machines and temporal logic. These specialized formalisms can be formalized in $\mathcal{L}$, stored into libraries, and used to define part of spec_0. For instance, the specification of an operating system could use finite state machines to describe the life cycle of processes and threads.

spec_0 may use multiple specialized formalisms to impose different constraints on the target programs. For instance, the specification of the operating system mentioned in the previous paragraph could also use temporal logic to express scheduling constraints. The logic of $\mathcal{L}$ provides semantic integration of the different specialized formalisms.

These specialized formalisms could be full-fledged specification languages, like the one in [9].[2] Thus, existing specifications in these languages can be used in derivations that are carried out using the refinement approach described in this paper.

Some specialized formalisms include composition mechanisms, e.g. parallel composition of state machines. These composition mechanisms can be also formalized in $\mathcal{L}$, and used in spec_0 as needed.

It is generally best to use the typical notations of a specialized formalism, e.g. graphs or tables for finite state machines and temporal operators for temporal logic. Extensible syntax in the theorem prover (e.g. as in Isabelle/HOL) can mimic some specialized textual notations. Furthermore, automatic translators can turn specialized notations (e.g. graphical) into their representation in $\mathcal{L}$.

Inference systems for specialized formalisms (e.g. model checkers) generate results that can be represented as theorems in $\mathcal{L}$. These inference systems can be treated as oracles by the theorem prover, or they can be enhanced to generate proofs that can be independently checked by the theorem prover.

[2] Languages of this kind are often considered general-purpose, but they are more specialized than $\mathcal{L}$.

2.2.6 Validation

A requirement specification can be validated in similarly to the validation of programming language formalizations discussed in Section 2.1.8:

- State and prove expected theorems about spec_0 (and its auxiliary types and constants). For example, the abstract operating system model mentioned in Section 2.2.2 could be proved to never reach certain states, and to reach certain states only under certain conditions.
- If tests are available, convert them into logical formulas and prove that they follow from spec_0.
- If informal requirements are available, follow them as closely as possible, explicitly tracing, via comments, each part of spec_0 to well-defined parts of the informal requirements.
- If any formal requirements are available, convert them into $\mathcal{L}$ and either use them as part of spec_0 or prove that they follow from spec_0. This process may involve formalizing in $\mathcal{L}$ the formalisms in which the available formal requirements are expressed (see Section 2.2.5).

Inconsistent requirements cause spec_0 to be empty, i.e. no program satisfies all the requirements.[3] Deriving an implementation from spec_0 (as described in Sections 2.3 and 2.4) provides a constructive proof of the consistency of the requirements. In some cases, it may be possible to prove, non-constructively, the non-emptiness of spec_0 before deriving the implementation, as part of validating the requirements.

2.3 Refine the Specification Stepwise

2.3.1 One-Step Refinement

Given a requirement specification spec_0, in principle it is possible to write a program in (the formalized subset of) the target programming language, convert it into the corresponding $\mathcal{L}$ term of type Prog, and prove that it satisfies spec_0.

This one-step refinement may work well for relatively simple requirements and programs. But as complexity grows, the proof must be manually decomposed into intermediate theorems and accompanying definitions, e.g. for invariants of the program. This may quickly become very laborious.

2.3.2 Multi-Step Refinement

Deriving a program by refinement is a way to decompose the proof that the program satisfies the requirements into simpler proofs (one for each refinement

[3]Inconsistent requirements, by themselves, do not cause inconsistencies in $\mathcal{L}$.

step), constructing the program along the way. This is done by defining predicates $\mathsf{spec}_1, \ldots, \mathsf{spec}_n$ that, with spec_0, form a monotonically decreasing sequence, i.e. for each i the following is a theorem in $\mathcal{L}$:[4]

$$\forall\, \mathsf{p} \cdot \mathsf{spec}_i(\mathsf{p}) \Rightarrow \mathsf{spec}_{i-1}(\mathsf{p}).$$

For some i, the inclusion may not be strict, i.e. spec_i and spec_{i-1} may be logically equivalent; however, spec_i generally has a form that is closer to the implementation than spec_{i-1}.

Like spec_0, the definition of each refinement spec_i is normally built out of auxiliary types and constants. Often, these bear some formal correspondence with the auxiliary types and constants of spec_{i-1}, and the proof of the inclusion of spec_i into spec_{i-1} hinges upon that formal correspondence.

2.3.3 Target-Independent Refinements

Some refinements are independent from the choice of the target program and language. For example, the abstract operating system model mentioned in Section 2.2.2 may use a record type to model the abstract state of the operating system. A refinement step may extend this record type with fields whose values can be always derived from the values of the other fields, but are cached for more efficient access—state-changing system calls must be accordingly refined to maintain this invariant relation between new and old fields. This refinement step can be used in the derivation of a C implementation as well as in the derivation of a Java simulator.

In general, given requirement specifications spec_0, spec'_0, spec''_0, ... with a common target-independent part, a sequence of m target-independent refinement steps can be shared by different derivations $(\mathsf{spec}_0, \mathsf{spec}_1, \ldots, \mathsf{spec}_n)$, $(\mathsf{spec}'_0, \mathsf{spec}'_1, \ldots, \mathsf{spec}'_{n'})$, $(\mathsf{spec}''_0, \mathsf{spec}''_1, \ldots, \mathsf{spec}''_{n''})$, ... (where $m < n$, $m < n'$, $m < n''$, ...). Since each of the refinements spec_j, spec'_j, spec''_j, ... (where $1 \leq j \leq m$) is a predicate over programs and therefore includes a target-dependent part, such refinements are not identical in general, but their significant (target-independent) parts are the same and usually determine their remaining (target-dependent) parts fairly mechanically.

2.3.4 Target-Dependent Refinements

Some refinements depend on the choice of the target program and language. For example, the refinement of the abstract operating system state described in Section 2.3.3 may be further refined by turning the record type into a C structure, in order to derive a C implementation. A different refinement may be used to derive a Java simulator, e.g. turning the record type into a class.

[4]It is assumed that $\mathcal{L}$ uses familiar notations for universal quantification, logical implication, and function application.

2.3.5 Specialized Formalisms

The specialized formalisms discussed in Section 2.2.5 can be useful not only to define part of spec_0, but also to define parts of the refinements spec_i. A specialized formalism can be used in spec_i even if it is not used in spec_{i-1}. For instance, spec_i may use a finite state machine to describe a certain piece of functionality more operationally than described in spec_{i-1}.

Some specialized formalisms are accompanied by their own notions of refinement. The formalizations in $\mathcal{L}$ of these specialized formalisms can be extended with formalizations of the accompanying notions of refinement, which can be used in derivations.

For example, if spec_{i-1} uses a finite state machine to describe a certain piece of functionality, spec_i could use a finite state machine that is a refinement of the former according to some notion of refinement between state machines, e.g. that every trace of the latter machine is also (or maps to) a trace of the former machine. Ultimately refinement amounts to spec_i logically implying spec_{i-1}, but the trace inclusion relation between the two finite state machines may the key reason why such implication holds.

The use of a specialized notion of refinement may or may not be appropriate. For example, the generation of a nonce for a cryptographic protocol[5] could be modeled via a specialized formalism that includes a non-deterministic process. Constraints on the probability distribution of the nonce may be imposed in $\mathcal{L}$, but outside the specialized formalism. Even though the specialized formalism may allow the non-deterministic process to be refined into a deterministic one that always returns the same nonce (defeating the security of the protocol), that would violate the probability distribution constraints and thus would not be a valid refinement in the sense of spec_i logically implying spec_{i-1}.

2.3.6 Automation

Given spec_{i-1}, its refinement spec_i can be manually written down (along with its auxiliary types and constants) and proved correct. This manual refinement style can be realized in existing theorem provers.

It is sometimes possible to generate spec_i automatically from spec_{i-1}, using an automated transformation. This automated refinement style can save effort and make derivations more robust against changes in the requirement specification.

For example, consider the refinement of the abstract operating system state discussed in Section 2.3.3. Using a manual style, this refinement could go as follow: (a) define the new record type of states S', which extends the old record type of states S with fields that cache values derivable from the others; (b) define two functions cache of type $\mathsf{S} \to \mathsf{S}'$ and trim of type $\mathsf{S}' \to \mathsf{S}$, that translate

[5]A nonce is a random number that should be used only once.

between S and S' (by caching and trimming the extra fields) and prove that they are each other's inverses; and (c) for each type and constant that depends (directly or indirectly) on S (spec_{i-1} being one such constant), define a new version of it that depends on S' instead (spec_i being the new version of spec_{i-1}) and prove that the new version is equivalent to the old version (in particular, that spec_i is logically equivalent to spec_{i-1}).

Using an automated style, the same refinement could go as follows: (a) and (b) are as above; but all the new types and constants of (c) are generated automatically from the old ones, along with the equivalence theorems, via an 'isomorphic type transformation'. This automated transformation works on any pair of isomorphic types (like S and S', but it is not limited to record types) with proven inverse isomorphisms (like cache and trim).[6]

This isomorphic type transformation is one of many automated transformations that are useful for refinement. Other examples are the application of rewrite rules, the application of algorithmic templates like 'divide and conquer', and the use of witness finding to construct terms (that refine other terms) [16].

Existing theorem provers do not provide this kind of automated transformations. Adding them to a theorem prover would make it more useful for refinement. The Specware tool [6] features a set of automated transformations (including some of the examples mentioned above) that operate on higher-order logic specifications.

2.3.7 Reuse

Parts of the refinements in a derivation may be more general than the system being refined. Thus, they can be stored into $\mathcal{L}$ libraries and used in multiple derivations. Prime examples are data type refinements.

2.4 Conclude with the Implementation

2.4.1 Last Refinement

The last refinement of a derivation must be a predicate that characterizes a unique program in explicit syntactic form. That is, $\mathsf{spec}_n(\mathsf{p})$ must be defined as $\mathsf{p} = \ldots$, where the ellipsis stands for a term of type Prog that contains only constants that formalize the syntax of programs—thus the term clearly corresponds to a program.

[6]The new generated functions may need to be optimized via further refinements. For instance, consider a state-changing function f of type $\mathsf{S} \to \mathsf{S}$, used in spec_{i-1}. The isomorphic type transformation may generate a new function f' of type $\mathsf{S}' \to \mathsf{S}'$ that is simply defined as $\mathsf{f}'(\mathsf{x}) = \mathsf{cache}(\mathsf{f}(\mathsf{trim}(\mathsf{x})))$. A further refinement step should optimize this definition to update the old fields in the same way as f does, and to update the extra fields directly, without using cache or trim.

The unique program that satisfies spec_n is the implementation obtained from the derivation. Because of the inclusion of each spec_i into spec_{i-1}, this program satisfies all the requirements expressed or implied by spec_0, functional as well as non-functional.

2.4.2 Code Generation

If the programs in Prog consist of abstract syntax, the term of type Prog used in the definition of spec_n represents the abstract syntax tree of the implementation. A code generator must convert this term into the corresponding program—more precisely, into program text whose abstract syntax tree is the one represented by the term, like a pretty printer. Errors in this code generator may lead to incorrect program text. However, the conversion is very simple.[7]

If the programs in Prog consist of concrete syntax, the term of type Prog used in the definition of spec_n represents the program text of the implementation. Thus, code generation is almost a no-op: it just copies program text into files.

Some theorem provers (e.g. Isabelle/HOL) include code generators that convert executable subsets of their logical language into functional programs. But no existing theorem prover includes a code generator of the kind described above. Adding this capability to a theorem prover would make it more useful for the approach to refinement described in this paper.

3 Future Work

3.1 Practicality

The approach to refinement described in this paper is doable in principle, but its practicality must be assessed. This applies not only to the overall approach, but also to some of the more specific techniques discussed above, e.g. the formalization and use of specialized formalisms and their accompanying notions of refinement (see Sections 2.2.5 and 2.3.5). Working through examples of increasing complexity is the best way to make this practicality assessment.

3.2 Theorem Prover Extensions

Sections 2.3.6 and 2.4.2 mention extensions that would make theorem provers more useful for the approach to refinement described in this paper. While working through examples as mentioned in Section 3.1, one or more theorem

[7] At least in comparison with typical code generators, which translate a language into a different one, and whose correctness involves the semantics of the two languages and their relation. In contrast, the code generator for terms of type Prog is purely syntactic.

provers could be extended with the needed automated transformations and code generators.

3.3 Research Topics

This paper touches upon issues that are research topics in their own right, independently from the approach to refinement described here. An example is the formalization of non-functional requirements discussed in Sections 2.1.5 and 2.2.4. Nonetheless, the refinement approach proposed here provides a good context and motivation to explore these topics.

3.4 This Paper

This paper should be extended with examples and with more bibliographic references.

References

[1] Common Object Request Broker Architecture (CORBA). www.omg.org/spec/CORBA.

[2] Coq. coq.inria.fr.

[3] Isabelle. www.cl.cam.ac.uk/research/hvg/isabelle.

[4] Java Native Interface. docs.oracle.com/javase/7/docs/technotes/guides/jni.

[5] The LLVM compiler infrastructure. llvm.org.

[6] Specware. www.specware.org.

[7] Stateflow. www.mathworks.com/products/stateflow.

[8] *IEEE Standard 1076-2008: VHDL Language Reference Manual*, January 2009.

[9] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[10] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Springer, second edition, 2002.

[11] Edsger W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8(3):174–186, September 1968.

[12] Cordell Green. *The Application of Theorem Proving to Question-Answering Systems.* PhD thesis, Stanford University, 1969.

[13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification – Java SE 7 Edition.* Oracle, February 2012.

[14] John McCarthy. A basis for a mathematical theory of computation. In Paul Braffort and David Hirshberg, editors, *Computer Programming and Formal Systems.* North-Holland, 1963.

[15] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, Revised Edition.* MIT Press, May 1997.

[16] Douglas R. Smith. Mechanizing the development of software. In Manfred Broy, editor, *Calculational System Design, Proceedings of the Marktoberdorf Summer School.* IOS Press, 1999.

[17] Richard J. Waldinger. *Constructing Programs Automatically Using Theorem Proving.* PhD thesis, Carnegie-Mellon University, 1969.

[18] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.