Requirement Capturing and 3CR Approach

Sergey Baranov, Vsevolod Kotlyarov, Motorola, GSG-Russia
Sergey.Baranov@motorola.com, Vsevolod.Kotlyarov@motorola.com

Julia Kapitonova, Alexander Letichevsky, Vladislav Volkov, ISS, Ukraine
Julia_Kapitonova@iss.org.ua, let@iss.org.ua, vlad@iss.org.ua

Abstract
An approach to requirement capturing based on formal methods is described with preliminary results of its application to particular industrial projects. The respective software ADE tool has been developed and is under piloting.

1. Introduction

The goal of 3CR (Checking Consistency and Completeness of Requirements) approach [4] is to benefit from using formal methods based on algebraic programming in requirement capturing for software projects. These methods are being integrated into software development technologies used in Motorola in form of tools for checking consistency and completeness requirements specifications expressed in MSC (Message Sequencing Charts). The benefits come from reducing the development cycle time and increasing the quality of the product. The tools comply with existing technologies supported by Telelogic and Test Automation Toolset [5].

The starting point of the project is the typical time distribution in the software development cycle: 40% is requirement capturing, 20% coding, and 40% testing. For requirement capturing this time includes not only the development of requirements but also their correction and refinement during the whole life cycle. The quality and minuteness of the decisions made at the phase of requirement capturing have a big impact on all the subsequent phases of software or hardware development. Therefore, initially consistent and complete requirement specifications for a system make it possible to avoid numerous errors at the latest phases of system design and coding. Our intention is to provide 10-15% decrease of software development expenses with a given degree of correctness in comparison to manual inspection of specifications. A certain reuse of already checked specifications may also contribute to this decrease.

2. Approach

3CR is based on the use of formal requirement languages adjusted to subject domains. Specialization is essential because it facilitates the inevitable transition from non-formal requirements to formal ones. However, the initial specifications are supposed to be prepared in a widespread engineering language like MSC with tools like Telelogic and then automatically translated in a formalized representation based on predicate and temporal logics.

The approach supposes verification of requirements independently of their implementations using an internal automatic proof system, which stays “invisible” for the engineers. Extended MSC is used for the description of transition properties of a system (interaction of system components inserted into specific environment) and a simplified logical language with modalities is used for the description of dynamic properties (like safety or liveness conditions) of a system under design. Formulations of these properties may be well re-used from domain libraries.

We specialize our approach to the following three classes of systems.
1. Embedded operating systems – asynchronous sequential multi-agent systems with interruptions and interaction with the external environment.

---

1 ISS is a subcontractor to Motorola, located in Kiev, the Ukraine
3. Tool

The 3CR tool ADE (Algebraic Design Environment) developed within this approach is an interactive open system, which can be adapted to various subject domains, including the three mentioned above. It provides the following features for development, verification, and validation of requirements for interactive systems:

1. Proving internal consistency and completeness of static requirements (descriptions of transition properties) expressed in MSC or SDL.

2. Proving dynamic properties of a system defined by static requirements including safety, liveness, and integrity conditions, which are expressed in a particular logical formalism.

3. Generating of executable specifications represented in a formal requirement language for a system defined by means of static requirements and simulating these models in user defined environments.

4. Generating complete test cases for executable specification of a system defined by verified requirement specifications and validating the implementations of the system defined in MSC or SDL.

5. Interface with Telelogic and TAT at all the phases of the design process.

All ADE features can be used in the automatic as well as in the interactive mode. The latter allows the user to interactively interpret warnings, which are generated by ADE, if the tool fails to prove certain properties, and helps to better locate inconsistent or incomplete requirements, which cause the problem. To study and solve the problems of consistency and completeness of requirements for interactive systems the theory of interaction of agents and environments [2] was used. The Action Language, developed on the basis of this theory, was implemented in the algebraic programming system APS [1], which served as the initial prototype for ADE.

As mathematical problems, the problems of consistency and completeness of requirements are well known to be algorithmically unsolvable even for the first order predicate calculus. (It’s a non-trivial task to just formulate these properties mathematically). To overcome this difficulty we have studied a number of real-life requirements used in the specific subject domains and based on their common specifics developed methods of proving sufficient conditions of their consistency and completeness. We arrived to the following classification of requirement statements and their logical structure.

1. Descriptions of observable entities (attributes) of a system and external environment, their types and parameters. All these descriptions are similar to the name and function definitions in programming languages and we can use different styles referring to the languages used in industry for requirement representation (SDL, UML, MSC).

2. General notions and their properties as predicates and functions used in requirements. A number of predefined properties to be proved are provided to the end-user in form of a menu or a library.

3. Static properties of the external environment formulated in a logical language, observable attributes of environment, and auxiliary notions introduced for this purpose.

4. Static requirements or the rules for system transitions. They describe the change of observable internal attributes of a system forced by the change of external attributes, events, or actions. A static requirement includes preconditions, which describe the possible states of the system before the transition, an action or event forcing the transition, and post-conditions describing possible states of the system after transition.

5. Dynamic properties expressing the requirements for the general behavior of the system. These requirements describe such properties as safety, liveness, or integrity conditions and are expressed in a simple logical language with temporal modalities (always, sometimes, eventually, etc.).

The major components of ADE are shown below (the proof system was inherited from [3]):

<table>
<thead>
<tr>
<th>PROOF SYSTEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reader of formalized requirements</td>
</tr>
<tr>
<td>Checker of static requirements</td>
</tr>
<tr>
<td>Prover of logical statements</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTERFACE WITH TELELOGIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERATOR OF EXECUTABLE SPECIFICATIONS</td>
</tr>
<tr>
<td>SIMULATING SYSTEM</td>
</tr>
<tr>
<td>VALIDATING SYSTEM</td>
</tr>
<tr>
<td>Test Generator</td>
</tr>
<tr>
<td>Validator</td>
</tr>
</tbody>
</table>
The reader accepts the input text reading it statement-by-statement and filling the corresponding data structures and ADE knowledge base with appropriate checking for syntactical correctness and preprocessing if needed. Static requirements are sent to the checker responsible for checking their internal consistency and completeness. The checker analyzes a requirement statement and generates a statement expressing the consistency of the current requirement with the ones accepted before, or the statement of completeness when all static requirements are accepted. Then the statement is sent to the prover for searching a proof. Three answers are possible to be received from the prover: proved, not proved, or unknown. The first means that consistency or completeness statement is proved. The second means that the statement is not provable, i.e., requirements are inconsistent or incomplete or there is lack of information to find the proof. The third answer means that the restrictions on the depth of a proof tree are not sufficient to find the proof. All results are gathered in a special verdict file along with the corresponding trace to the initial requirements and can be viewed by the user for making further decisions. The user can also intervene into the process of checking consistency and completeness on different levels of consideration.

Dynamic properties are checked after accepting all static requirements addressing the prover directly. They are logical statements expressing the properties of a system in terms of a predicate calculus language, extended with temporal modalities, higher order functions, and types. If an inductive proof is needed, all static requirements are used for generating lemmas to prove the inductive step. Direct addressing the prover is possible also when some statement expressing the properties of external environment or auxiliary notions is checked to be a consequence of already made assumptions.

After checking consistency and completeness of static requirements they are used for automatic generation of an executable specification of the system, which satisfies the static requirements. Dynamic requirements are already proved to be consequences of the static ones, so the system also satisfies dynamic requirements.

The next step of the system design is the use of obtained information for the next stages of the development. Executable specifications can be used for generating complete test cases used for testing the system. After coding the model of the system may be validated with ADE.

4. Results

Four examples from industrial applications were specified and checked for consistency and completeness with ADE. Each represents a corresponding class of systems. Formal specifications for the kernel of the operating system OSEK were considered as an example of an asynchronous sequential system. The bus protocol for processor element was an example of requirements for a synchronous agent inserted into a parallel environment. And we also used two examples of parallel environments, one for a communication protocol in a multiprocessor system, and another for a telecommunication protocol.

The table below gives quantitative characteristics of typical projects in these domains.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Number of requirements</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous sequential environments</td>
<td>49</td>
<td>10 min</td>
</tr>
<tr>
<td>Synchronous sequential agent inserted into a parallel environment</td>
<td>26</td>
<td>4.5 min</td>
</tr>
<tr>
<td>Parallel asynchronous environment</td>
<td>20</td>
<td>3 min</td>
</tr>
<tr>
<td></td>
<td>46</td>
<td>1h 7 min</td>
</tr>
</tbody>
</table>

5. References


Motorola is a trademark registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.