Title: AN INTEROPERABLE EXTENSIBLE SYSTEM FOR THE GENERATION OF VERIFIED SOFTWARE CODE

Abstract: A system for generating verified software code from an original model of a system under development (SUD) is described, where the original model expresses the SUD in terms of components, structural behaviour between components (and overall behaviour of the SUD). The system comprises: a plurality of modelling language adaptors arranged in parallel to receive the original model of the SUD and to convert the original model into a transformed model expressed in a common language, each modelling language adaptor being configured to convert an original model written in a specific modelling language into the transformed model expressed in the common language such that the system can generate verified code from the original model written in any of a plurality of different modelling languages; a model verifier for mathematically verifying if the transformed model satisfies at least one of its specified requirements and if the transformed model does not, generating verification analysis data expressing how the at least one specified...
SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TI, TM, TN,
TR, TT, TZ, UA, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States (unless otherwise indicated, for every
kind of regional protection available): ARIP0 (BW, GH,
GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ,
UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ,
TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK,
EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV,
MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM,
TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW,
KM, ML, MR, NE, SN, TD, TG).

Published:
— with international search report (Art. 21(3))

...has
mathematically verified the transformed model as satisfying the at least one of its specified requirements.
AN INTEROPERABLE EXTENSIBLE SYSTEM FOR THE GENERATION OF VERIFIED SOFTWARE CODE

TECHNICAL FIELD

The present disclosure relates to an interoperable extensible system for the generation of verified software. More particularly though not exclusively, the present invention relates to a system for and a method of automatically generating verified executable code from a model of system under development (SUD) regardless of which language the model is written in. The system is extensible to enable the handling of complex models which result in industrial scale code generation (millions of lines of code).

BACKGROUND

High cost of complex software development

Today's products across sectors depend increasingly on software to function, for example in aerospace, manufacturing, defence and automotive. Embedded software systems are rapidly increasing in scale and complexity to meet this need. Factors driving this include requirements for increased automation and autonomy, improved reliability and safety, increased connectivity, and increased performance. To manage this complexity, systems are becoming more distributed, and are using more complex hardware. This increase in software complexity and scale has led to the cost of software development rising rapidly. Traditional testing practices are not able to tackle this challenge alone; in current practices, it is already common for testing and rework to take up to 50% of project costs. In a number of sectors, for example aerospace, it is recognised that without game changing innovation, software will be a barrier for future innovation of advanced systems and architectures, rather than the enabler that software should be.

Model-based engineering

The trend over the last decade has been towards using model-based engineering in an attempt to tackle the increasing cost and complexity of software. Model-based software engineering (MBE) is a software development process in which all or parts of a system under development (SUD) are expressed in a modelling language in the form of a model. This typically includes aspects such as its intended behaviour, performance characteristics, its decomposition into components, and the structural and behavioural relationships between them. Multi-disciplinary teams of engineers make the model 'by hand' using a modelling tool. Although model-based engineering processes and tools have provided some beneficial improvements, in particular by reducing the amount of information being captured on paper (i.e. replacing this with the models) and enabling automation of certain aspects of the software development process (for example, code generation). However, on its own, model-based engineering is not able to tackle the root causes of the escalation of cost and complexity, since currently practices still rely mainly on traditional testing practices.
Thus, the occurrence and cost of errors propagating through the software development process and into products remain largely unchanged by adopting a model-based engineering approach alone.

Modelling languages
Modelling languages can be categorised as being textual or graphical, proprietary or open-standards based, general purpose or domain specific. Each proprietary modelling language is typically used with a specific modelling tool. Adopting a proprietary modelling language also means adopting the modelling tool specific to that language. This can increase the problem of so called 'vendor lock-in', where an end-user has their model expressed in a proprietary modelling language supported by a specific tool (for manipulating the model) supplied by a single vendor, and is unable to move to another tool supplier. Models expressed in a proprietary modelling language also typically make interchangeability between different modelling languages and tools more difficult. Proprietary modelling languages that are currently used across numerous industries include ASD and Simulink. Such languages require specialised training in order to be used, and also lock users into using tools which are compatible with that language being typically from a single vendor (source).

An open standards-based modelling language is one defined by an industry-recognised standards body. UML and various derived modelling languages such as SysML and MARTE are examples of graphical modelling languages and are among the most widely adopted in industry and are supported by many commercial and open-source modelling tools. Domain specific modelling languages (DSL) tend to be specific to a given development organisation and are designed to express easily concepts and abstractions specific to that organisation and the types of systems it develops. Most DSLs are proprietary and therefore bound to a specific modelling tool; others, such as MARTE, are defined as specialisations of general purpose modelling languages such as UML.

Poor interoperability between modelling languages and modelling tools
The function of a modelling tool is to enable users to manipulate models namely to create, edit, visualise, store and retrieve models expressed in a given modelling language. Different modelling languages and tools are needed to tackle different aspects of software and its development. These are mostly point solutions, making seamless integration between them extremely challenging and error prone. Generally, a given modelling tool supports only one modelling language. In other words, such tools and models are often not interoperable, which means models created using one tool can rarely be used with another tool, and the user is locked-in to using a single tool. Where proprietary modelling languages from a given source are used, interoperability is impeded because most of the tools available for that model are provided by the source and as such they only support the modelling language of that source. Furthermore, users sometimes develop customised bespoke specialist tools that tie together specific existing tools in order to circumvent this tool lock-in problem
described above. This can be inefficient in practice by creating a significant amount of work that is orthogonal to their core business, requiring a disproportionate amount of time and money to be invested in skills, tool development and maintenance. Furthermore, the precise manner in which these bespoke tools operate are not publicised as they are typically internal to the business.

Where models are expressed in an open standards-based modelling language, interoperability is impeded by the ambiguous semantics of many modelling languages and the fact that a given model does not necessarily have same meaning in every tool. For example, the same UML model described in two different commercial modelling tools, such as IBM Rhapsody and MagicDraw, will each have their own interpretation of the semantics. These will be different from each other and therefore the meaning of the same UML model in the two tools will also differ.

A corresponding issue to the interoperability of models with different tools, is that the generated executable code that results from each tool is often subtly different, which means that there is still a significant burden of manual work and testing in order to integrate the resulting code. This means that the challenges of cost and complexity are still present. The current state of the art is, therefore, such that interoperability and model exchange between modelling tools varies from poor to non-existent. This has a direct impact on the ability to automatically generate code that can be immediately run on the target system from heterogeneous models as described below.

**Lack of automation**

Many existing modelling languages either lack a precise formal semantics entirely or have a poorly defined semantics that are ambiguous in practice. Without a proper semantics, there is little or no guarantee of correspondence between the model and the executable code. The lack of a precise semantics also impedes the use of automated features such as formal verification technologies, executable code generation, automated test-case generation and simulation. Such technologies depend on a single semantics common to them all in order to enable interoperability. For example, if there is no single semantics, each code generator has its own interpretation and generates different code, which makes it inherently difficult to switch between different code generators.

Most MBE approaches also lack significant automation, making them labour intensive, costly and error prone with limited automation. For example, most models are transformed to executable code by hand. If the code is not automatically generated from the model, or if the generated code has to be extensively modified by hand, then the models and the code, over time, diverge. This means that there would no longer be any single source of truth about how the system behaves, making maintenance dramatically more expensive and decreasing the amount of code reuse between different projects.
Lack of formal verification

Model-based engineering tools, even those which have code generation, still rely entirely on conventional testing practices to ensure the implementations are high-quality and meet the requirements. Instead, such errors are only revealed through testing the executable code, meaning that errors are found late during the development process, when such errors are expensive to detect and remove. This requires large sets of test cases to be developed and repeatedly executed as errors are discovered and changes made to repair them.

Testing can only reveal defects and not prove their absence, so complex systems that are assessed through testing only may be unreliable. For example, it is widely known that there is no feasible amount of testing that can demonstrate the correctness of event driven, reactive and concurrent systems are due to the systems being inherently nondeterministic. This nondeterminism means that a given test case only covers a very small amount of the possible behaviour of the system. Further, since the systems are nondeterministic it is infeasible to construct test cases that guarantee to cover certain cases, since the nondeterministic parts of the system cannot be controlled by the test case, by definition. This is becoming an increasing problem as software is becoming increasingly concurrent and also ever more complex. Testing alone is simply unable to cope with this development.

In addition, traditional testing techniques can only be used to test the actual executable code. This means that errors made in the requirements and design phases remain undetected until late in the development process which raises the cost of removing them.

Tools that combine formal verification and code generation

Over the last decade there has been a significant increase in the number of tools that attempt to address some of the above problems. For example, tools have been developed that combine model-based development with partial or complete code generation. Others have sought to enable the integration of code generation and formal verification for simplified proprietary languages and highly restricted system architectures. However, there has not been any attempt to combine formal verification and code generation to enable interoperability between different modelling languages and thus different tools, or general system architectures. Presently-available tools that combine formal verification and code generation include ASD, IBM Rhapsody, SCADE and SLDV.

ASD, for example described in WO 2005/106649, allows users to specify simple, single-component models. It can then verify that the component model is correct with respect to its interface specification, under restricted assumptions about the runtime semantics and system architecture that the component can operate in. This tool can also generate source code. ASD models are expressed in a restricted proprietary modelling language, thus creating a significant barrier to adoption: users have to learn
and adopt a new language, a new tool and a new way of working, in order to benefit
from this tool. Furthermore, legacy models created in existing modelling languages
which are different to ASD have to be rewritten in the new restricted proprietary
modelling language in order to be able to be used. The ASD modelling language is
restricted to expressing the interface and implementations of individual components; it
cannot express system-level abstractions or concepts, and cannot express functional
or non-functional system requirements. Non-functional requirements specify criteria
that can be used to judge the operation of a system, rather than specific behaviours of
the system; examples of non-functional system requirements are performance,
scalability, and reliability. In contrast, functional requirements define what a system is
supposed to accomplish in terms of the specific behaviour or functions of the system.
For example, these can be captured in terms of sets of input, behaviour, and outputs.

Consequently, ASD cannot verify conformance between component designs and
system requirements nor can it verify system properties. The ASD modelling language
also only supports simplified architectures that are compositional and arranged in a
tree. ASD does not have its own independent executable semantics; instead it is
derived from the process algebra CSP (Communicating Sequential Processes) used
for the formal verification. Thus, there is not a single point of truth from which all
models, abstractions and source code are generated from, making ASD difficult to
extend. Most notably, this makes it difficult to extend ASD to use different verification
tools.

IBM Rhapsody is a model-based development tool that includes extensive support for
automated code generation from SysML and UML models. Several attempts have
been made at integrating formal verification into Rhapsody, but all of these attempts
are specific to Rhapsody semantics which are not compliant with any standards, and
thus are not independent of the modelling language or tool. Furthermore, none of
these attempts have yet proven to scale to industrial software.

Safety Critical Application Environment (SCADE) is both a proprietary modelling
language and a modelling tool. SCADE is based on the theory of synchronous
languages for deterministic real-time systems and in particular on the Lustre and
Esterel languages for modelling dataflow and control flow, respectively. SCADE
models are restricted to a synchronous, cycle-based execution model, the common
sampling/control model from control engineering, in which a continuous loop executes
alternating between reading input sensor and executing application actions. On each
cycle, input sensors are read and cycle outputs are computed and fed back to the
environment being controlled. SCADE does not support other more general execution
models for reactive, event driven, concurrent systems; it does not support modelling
the separation of interface and implementation; and it cannot express nondeterminism.
SCADE supports simulation, formal functional and timing verification and code
generation for C and Ada.
SLDV is capable of both verifying properties about Simulink models, and also automatically generating test cases. It integrates in with code generators provided by the same vendor and thus allows the system to be verified and for code to be generated. Since SLDV supports only Simulink, it is not independent of either the modelling language or the modelling tool and therefore does not enable interoperability. Since Simulink has no independent formal semantics, it would be impossible to translate another modelling language into it.

The current method of software development is now described in greater detail with reference to Figures 1 and 2.

Referring to Figure 1, there is shown a plot 100 of the relative cost 110 of fixing software defects at different stages 120 of the software development process e.g. requirements 130, design 140, coding 150, testing 160 or operation 170. From requirements 130 to coding 150, the relative cost of fixing defects remains relatively low. Improved testing procedures can help in the software development process; however testing 160 can only reveal defects and cannot prove their absence. This means that defects are often found late in the development process and such defects are, as illustrated in Figure 1, extremely expensive to fix, both financially and in terms of their impact to the project's timeline and time-to-market. The cost of fixing software defects is the highest during the operation stage 170.

Figure 2 depicts a standard model-based software development process 200. There are several approaches to software development which have been described previously, all with various problems that contribute towards escalating costs and increasing difficulty in managing complexity and ensuring reliability.

The core general advantage of MBE is that it can improve the quality of software as it enables requirements to be specified early in the design process and allows these requirements to be traced from design through to implementation. It also improves communication between stakeholders and the software development team, reducing the scope for costly misunderstandings in requirements. MBE also opens up the possibility of reusing components between different projects, as the precise specifications for each component are accurately captured by the model.

A typical prior art MBE process, as shown in Figure 2, comprises several distinct phases, each of which adds information to the model. Every development organisation that adopts MBE has its own version of the following process and makes its own choice of modelling language and modelling tools.

Typically, the following phases can be identified:

Requirements Analysis 210: Engineers who are experts in the problem domain gather and analyse information from stakeholders to determine the system requirements of
the SUD. Collectively, the system requirements specify the functional and non-functional operational requirements the design intent of the SUD in terms of the concepts and abstractions appropriate to the problem domain. They are expressed in the chosen modelling language, and added to the model using the chosen modelling tool.

**System Design 220:** Engineers who are experts in the solution domain, analyse the system requirements and determine an appropriate system design for the SUD. This includes decomposing the SUD into appropriate subsystems and components, allocating responsibilities for the system requirements between them and identifying interfaces between components. The system design is specified in terms of concepts and abstractions appropriate to the solution domain. It is expressed in the chosen modelling language and added to the model using the chosen modelling tool.

**Component Design 230:** In this phase, software engineers specify the design of the components to the level of detail necessary to enable their implementation in source code. The detailed designs are expressed in the chosen modelling language and added to the model using the chosen modelling tool.

**Implementation 240:** Using the component designs, software engineers implement the individual components in the chosen programming language. Depending on the chosen modelling language and modelling tool, the components are created by manual programming, automated code generation tools, or a combination of both.

**Unit Testing 250:** During the component design phase, tests would have been developed for the individual components in isolation. During the unit testing phase, test engineers implement these tests as test programs and use these to check if the components are correct, in isolation. Depending on the chosen modelling language and modelling tool, implementing the components and test cases is performed by manual programming, automated code generation tools, or a combination of both.

**Integration Testing 260:** Once the individual components have been tested, larger components are combined together and then tested in combination. This verifies if the components can correctly operate in conjunction with each other. This phase can reveal errors in any of the earlier phases and as a consequence can require rework of software design 220, 230 and subsequent implementation 240.

**System Testing 270:** In this phase, the system components are combined to form the SUD and tested together with any other system with which the SUD interacts in the operational environment.

**Acceptance Testing 280:** In this phase, a validation team performs end-to-end testing to validate the system to ensure full compliance of the implementation with the system requirements.
Errors 290 generated by the testing relate to requirements, design and implementation and these errors are fed back to the MBE platforms A and B as appropriate.

The present invention has been devised to mitigate or overcome at least some of the above-mentioned problems.

SUMMARY OF THE INVENTION

The present invention is directed to an interoperable extensible system arranged to receive a model, to formally verify the model, and to enable the correction of the model for use in creating executable code. The system comprises one or more modelling language adaptors comprising a model transformer arranged to transform a received original model expressed as a functional model specification (defined later) into a transformed model. The system also comprises one or more formal verification tools arranged to receive the transformed model; to verify mathematically if the transformed model satisfies its specific requirements; and in some cases to create an analysis output. The transformed verified model can then be automatically converted into an executable code.

An embodiment of the present invention, called the 'Coco System', is easily accessible and usable by engineers skilled in model-based engineering practices in industrial contexts because it allows them to build models using modelling languages and tools that are already familiar to them. This requires full automation across all aspects of the technology stack and the ability for all the backend technologies to be fully hidden and seamlessly integrated into frontend modelling tool being used by the engineers. This non-invasive aspect of the Coco System overcomes the barriers of adoption faced by some of the prior art tools, such as ASD, that requires users to learn a new language, tool and engineering paradigm.

The Coco System is modelling language independent, thereby enabling integration with and interoperability between different modelling languages ranging from open standards, such as UML and SysML, to proprietary languages.

Embodiments of the present invention, seek to tackle the scale of increasing software complexity and resulting escalation of costs by providing unity between model-based engineering, formal verification, code generation, and testing.

The present inventors have realised that when used in conjunction with a model-based development tool, formal verification can provide the essential guarantees that the resulting code is being developed from correct verified designs. Furthermore, advantageously it enables development of automated tools for generating source code that accurately reflects the designs and requirements. It is infeasible to achieve this manually for complex industrial software systems.
The Coco System combines scalable formal verification, model-based engineering and code generation in an innovative way. The system takes input in one or more model-based languages and is able to both formally verify if the model is correct, and generate equivalent executable code. The system can be fully integrated into existing industrial model-based engineering platforms and software development processes, thereby removing the barriers of adoption that have proven to be so problematic to date. Furthermore, it enables interoperability between model-based languages and model-based development tools, a key aspect that is yet another barrier in the adoption of such model-based technologies in industry at present.

More specifically, according to a first aspect of the present invention there is provided a system for generating verified software code from an original model of a system under development (SUD). The original model expressing the SUD in terms of components, structural behaviour between components, and overall behaviour of the SUD, the system comprising: a plurality of modelling language adaptors arranged in parallel to receive the original model of the SUD and to convert the original model into a transformed model expressed in a common language, each modelling language adaptor being configured to convert an original model written in a specific modelling language into the transformed model expressed in the common language such that the system can generate verified code from the original model written in any of a plurality of different modelling languages; a model verifier for mathematically verifying if the transformed model satisfies at least one of its specified requirements and if the transformed model does not, generating verification analysis data expressing how the at least one specified requirement is not met; and a code generator for generating executable code from the transformed model once the model verifier has verified the transformed model as satisfying the at least one of its specified requirements.

The present invention permits the models written in any one of a variety of different languages to be formally verified and is also able to automate the transformation from model to executable code. Furthermore, the present invention enables the use of formal verification tools to mathematically prove the correctness of software and consequently to reduce the need for testing, regardless of the language or tools used to create the original model. The remaining testing advantageously becomes more effective because the resulting code comprises fewer undiscovered errors.

The original model can be written to describe both an expected behaviour of the SUD and an actual behaviour of the SUD using conventional functional model specifications. In this case, each of the plurality of adaptors can be arranged to generate the transformed model by converting the conventional functional model specifications into a processing language comprising computational processing commands and control function commands. The computational processing and control function commands in combination are functionally equivalent to the conventional functional model specifications.
In this way, complex functional model specifications defining expected and actual behaviour of the SUD can be transformed into a common processing language form which advantageously enables many different types and forms of behaviours to be expressed concisely and accurately. The further advantage of this transformation is that different architectures of the SUD some of which require incredibly complex functional model specifications (such as large state machines) can be expressed without loss of functionality and in a manner which enables optimisation for code verification and generation.

The plurality of adaptors may be arranged to convert the conventional model specifications into the processing language including commands which describe the structure of the SUD. Using an expressive common processing language provides many different types of structures to be described efficiently.

The plurality of adaptors may be arranged to generate commands which describe the SUD as a plurality of components, namely elements of the SUD and how they operate together.

The transformed model can be arranged to express the original model in terms of components, having port constructs and interface constructs: a component representing an element of the SUD or a collection of such elements and include a description of the behaviour of the component including the component's interaction with other components and one or more port constructs; the port constructs being specific interaction points in a component; and include an associated behaviour defining allowed interactions via the port construct; and the interface constructs defining the functions available on each interface of the component to interact with another component and notification signals that a component can produce to communicate with other components.

The plurality of adaptors can be arranged to generate commands which capture verifiable behaviour of a component or a port construct in terms of the relationship between the inputs and outputs of that component or port construct.

The verifiable behaviour may be in the form of a state machine.

The plurality of adaptors may be arranged to generate commands which describe communications between components of the SUD.

Also the plurality of adaptors may be arranged to generate commands which express the communications in the original model in terms of signals, and/or function calls and function responses.
The expected behaviour may relate to one or more components of the system that represents a discrete part of the SUD.

The model verifier can be arranged to compare the actual behaviour of the one or more components of the transformed model with the expected behaviour of the one or more components of the transformed model to verify the transformed model.

The actual behaviour of the one or more components of the transformed model is processed in parallel to the expected behaviour of the one or more components of the transformed model. Having both expected and actual behaviour advantageously enables faster verification as these components can be processed independently.

The one or more components may comprise an internal component representing a component of the SUD and an external component representing a component outside the SUD which interfaces with the SUD; and the verifier can correspondingly be arranged to verify the equivalence of the actual and the expected behaviour of the internal component using the expected behaviour of the external component.

The original model may describe system requirements at multiple different levels within the SUD using conventional functional model specifications and each of the plurality of adaptors may be arranged to generate the transformed model by converting the system requirements described by the conventional functional model specifications into verifiable system requirements expressed in the common processing language.

The model verifier may be arranged to compare one or more of the verifiable system requirements with the behaviour of the transformed model to determine if the one or more system requirements are met by the behaviour of the transformed model.

The plurality of adaptors can be arranged to create the transformed model by simplifying the computational processing and control function commands from the conventional model specifications. This simplification can speed up and make the verification process much more efficient and faster as well as improving code generation.

The model verifier may comprise an optimiser for optimising the common language of the transformed model. Such optimisation can also lead to faster and more efficient verification and/or code generation.

The model verifier may comprise a plurality of different model verifiers arranged to work in parallel, each model verifier being arranged to verify a different aspect of the transformed model. In this way the system can verify different aspects of the model of the SUD concurrently and use this in efficient and more accurate code generation.
Each of the plurality of adaptors can be arranged to generate the transformed model using computational processing and control function commands which are arranged to express non-determinism of at least part of the original model. This importantly enables the transformed model to express the expected behaviour of the SUD in a verifiable manner.

Each of the plurality of different modelling adaptors can comprise a language-specific model validator arranged to receive the original model and to validate the semantics of the original model in the language of the language of the original model and if the original model is validated, to continue processing the original model to convert the original model into the transformed model expressed in the common language.

Each of the plurality of modelling language adaptors may comprise a modelling language-specific analysis transformer for transforming verification analysis data output from the model verifier and expressed in the common language, into the specific modelling language associated with the modelling language adaptor. This advantageously enables feedback to be provided back to the creator of the original model in a form which is suitable to their domain, i.e. in a language in which the original model was written.

The common language may be a high-level language and the system may further comprise a high-to-low transformer for transforming the transformed model written in the high-level language output from one of the modelling language adaptors into a low-level language specification for input into the model verifier, wherein the high-to-low transformer inserts additional statements into the transformed model to make the execution semantics of the transformed model more specific.

The low-level language specification may be iteratively optimised to remove redundancy, for example by Boolean expression simplification, removal of variables which are not part of the verification process and/or coalescing states of components, before being input into the model verifier. Clearly this optimises the function of the model verifier and increases speed/efficiency of operation.

The system may further comprise a low-to-high transformer for transforming the low-level language specification output from the model verifier into a transformed model expressed in a high-level language for input into the modelling language adaptor, wherein the low-to-high transformer removes additional statements previously inserted by the high-to-low transformer into the transformed model.

A model verifier may comprise: a low-to-verification transformer for transforming the low-level language specification input into the model verifier into a model verification language specification, wherein the model verification language specification expresses the transformed model mathematically; and a model verification engine for carrying out a formal verification of the mathematically-expressed transformed model.
in the model verification language which is compatible with the model verification engine.

An example of such a model verification language is CSM and a CSM verifier engine would be used in the model verifier. Alternatively, another model verifier may comprise: a low-to-SMT transformer for transforming the low-level language specification input into the model verifier into a SMT specification, wherein the SMT specification expresses the transformed model mathematically in terms of logical equations; and an SMT solver for carrying out a formal verification of the mathematically expressed transformed model.

The system may further comprise a verification-to-low transformer for transforming the model verification analysis output from the model verification engine to a low-level language specification.

According to a second aspect of the present invention there is provided a method of generating verified software code from an original model of a system under development (SUD), the model expressing the SUD in terms of components, structural behaviour between components, and overall behaviour of the SUD, the method comprising: receiving the original model of the SUD at one of a plurality of modelling language adaptors and converting the original model into a transformed model expressed in a common language, the converting step comprising converting an original model written in a specific modelling language into a transformed model expressed in the common language such that verified code is generated from the original model written in any of a plurality of different modelling languages; mathematically verifying if the transformed model satisfies at least one of its specified requirements, and if it does not, generating verification analysis data expressing how the at least one specified requirement is not met; and generating executable code from the transformed model once the model verifier has mathematically verified the transformed model as satisfying the at least one of its specified requirements.

According to another aspect there is provided a system for formally verifying the correctness of an original model of a system under development (SUD), the original model expressing the SUD in terms of components, structural behaviour between components and both expected and actual behaviour of the SUD using functional model specifications, the system comprising: a modelling language adaptor arranged to receive the original model of the SUD and to convert the original model into a transformed model expressed in a processing language such that the system can use the transformed model to verify the original model; wherein the modelling language adaptor is arranged to generate the transformed model by converting the conventional functional model specifications into the processing language, the processing language comprising computational processing commands and control function commands, the computational processing and control function commands in combination being
functionally equivalent to the conventional functional model specifications; and a model verifier for mathematically verifying if the transformed model satisfies at least one of its specified requirements and if it does not, generating verification analysis data expressing how the at least one specified requirement is not met.

The system in this case may further comprise a code generator for generating executable code from the transformed model.

According to another aspect there is provided a system for automatically generating software code from an original model of a system under development (SUD), the original model expressing the SUD in terms of components, structural behaviour between components and both expected and actual behaviour of the SUD using functional model specifications, the system comprising: a modelling language adaptor arranged to receive the original model of the SUD and to convert the original model into a transformed model expressed in a processing language such that the system can use the transformed model to generate code corresponding to the original model; wherein the modelling language adaptor is arranged to generate the transformed model by converting the conventional functional model specifications into the processing language, the processing language comprising computational processing commands and control function commands, the computational processing and control function commands in combination being functionally equivalent to the conventional functional model specifications; and a code generator for generating executable code from the transformed model.

According to another aspect there is provided a system for formally verifying the correctness of an original model of a system under development (SUD), the original model expressing the SUD in terms of components, structural behaviour between components and overall behaviour of the SUD, the system comprising: a plurality of modelling language adaptors arranged in parallel to receive the original model of the SUD and to convert the original model into a transformed model expressed in a common language, each modelling language adaptor being configured to convert an original model written in a specific modelling language into the transformed model expressed in the common language such that the system can verify the original model written in any of a plurality of different modelling languages; a model verifier for mathematically verifying if the transformed model satisfies at least one of its specified requirements and if the transformed model does not, generating verification analysis data expressing how the at least one specified requirement is not met.

According to another aspect there is provided a system for automatically generating software code from an original model of a system under development (SUD), the original model expressing the SUD in terms of components, structural behaviour between components and overall behaviour of the SUD, the system comprising: a plurality of modelling language adaptors arranged in parallel to receive the original model of the SUD and to convert the original model into a transformed model.
expressed in a common language, each modelling language adaptor being configured to convert an original model written in a specific modelling language into the transformed model expressed in the common language such that the system can generate code from the original model written in any of a plurality of different modelling languages; and a code generator for generating executable code from the transformed model.

According to another aspect there is provided a computer-implemented method of formally verifying the correctness of an original model of a system under development (SUD), the original model expressing the SUD in terms of components, structural behaviour between components and both expected and actual behaviour of the SUD using functional model specifications, the method comprising: receiving the original model of the SUD at a modelling language adaptor; converting the original model into a transformed model expressed in a language such that the transformed model can be used to verify the original model; generating the transformed model by converting the conventional functional model specifications into the processing language, the processing language comprising computational processing commands and control function commands, the computational processing and control function commands in combination being functionally equivalent to the conventional functional model specifications; and mathematically verifying if the transformed model satisfies at least one of its specified requirements and if the transformed model does not, generating verification analysis data expressing how the at least one specified requirement is not met.

According to another aspect of the present invention there is provided a computer implemented method for automatically generating software code from an original model of a system under development (SUD), the original model expressing the SUD in terms of components, structural behaviour between components and both expected and actual behaviour of the SUD using functional model specifications, the method comprising: receiving the original model of the SUD at a modelling language adaptor; converting the original model into a transformed model expressed in a processing language such that the system can use the transformed model to generate code corresponding to the original model; generating the transformed model by converting the conventional functional model specifications into the processing language, the processing language comprising computational processing commands and control function commands, the computational processing and control function commands in combination being functionally equivalent to the conventional functional model specifications; and generating executable code from the transformed model.

According to another aspect there is provided a computer-implemented method of formally verifying the correctness of an original model of a system under development (SUD, the original model expressing the SUD in terms of components, structural behaviour between components and overall behaviour of the SUD, the method comprising: receiving the original model of the SUD at one of a plurality of different
modelling language adaptors arranged in parallel; converting the original model into a transformed model expressed in a common language, the converting step comprising converting an original model written in a modelling language specific to one of the modelling language adaptors into a transformed model expressed in the common language such that the original model written in any of a plurality of different modelling languages can be verified; mathematically verifying if the transformed model satisfies at least one of its specified requirements, and if the transformed model does not, generating verification analysis data expressing how the at least one specified requirement is not met.

According to another aspect there is provided a computer-implemented method of generating verified software code from an original model of a system under development (SUD), the model expressing the SUD in terms of components, structural behaviour between components and overall behaviour of the SUD, the method comprising: receiving the original model of the SUD at one of a plurality of different modelling language adaptors arranged in parallel; converting the original model into a transformed model expressed in a common language, the converting step comprising converting an original model written in a modelling language specific to one of the modelling language adaptors into a transformed model expressed in the common language such that code can be generated from the original model written in any of a plurality of different modelling languages; and generating executable code from the transformed model.

In any of the above aspects of the present invention where a code generator is provided, the code generator may comprise a plurality of different code adaptors configured to operate in parallel, each code adaptor being arranged to generate and output a specific type of executable code, and may comprise a core code generator for converting the transformed model into the specific type of executable code for that adaptor.

Also the code generator may comprise a test case generator for generating individual test cases from the transformed model to determine if a component or a collection of components has its required behaviour, each component including a description of the behaviour of the component, the component's interaction with other components and one or more port constructs, and the output of the test case generator can be combined with the output of the core code generator to provide the executable code.

The code generator may comprise a conformance checker generator for verifying at run time if an external component with which the SUD interacts, is behaving according to its port construct, the output of the conformance generator being combined with the output of the core code generator to provide the executable code.

Finally, the code generator may comprise a legacy code adaptor for automatically generating code that binds an external component with legacy code, the output of the
legacy code adaptor being combined with the output of the core code generator to provide the executable code.

There are several other advantages which arise out of the present invention by itself and from a combination of the invention and various features described above. These advantages are expressed below.

A key advantage of the present invention is that the system provides assurance that the resulting executable code generated by the system correctly reflects the designs or requirements of the SUD. Software developed using conventional processes has very poor traceability, in that it is hard to understand how the high-level requirements identified during the design phase are implemented by the actual model and executable code. The problems of the prior art which the present invention and the different combinations of its various features, address include:

a) Certification and validation: it is difficult to determine how the software implements individual requirements. Particularly in safety critical and regulated domains, it is vital to be able to trace requirements that relate to safety properties and determine whether or not the implemented software satisfies them; existing techniques make this difficult. This significantly increases the cost of validating the software’s correctness and obtaining certification. This is a major cost driver in many safety-critical industries.

b) Limits product evolution: many large and complex software systems are intended to have a long operational life, during which many additions, functional enhancements and modifications will be made. Lack of traceability makes it difficult to determine the effect of changes on existing operational characteristics and performance. It also makes it difficult to determine the changes needed to implement new features or modify existing features.

c) Increased cost: making test cases is labour intensive, expensive and error prone, since requirements specifications and requirements models have to be read and understood by test engineers and the test cases designed and implemented.

d) Test time and cost increases more rapidly than complexity: an increase in complexity e.g. code size results in a disproportionate increase in the number of tests required to establish a given confidence level in software reliability and correctness. For large and complex systems, the testing time and costs consume more of the project schedule and budget than is used to develop the software being tested.

The Coco System, embodying the present invention, is able to automatically: verify the correctness of the system with respect to its requirements and specifications; generate
equivalent executable code that implements the SUD; and generate analysis feedback to users. Furthermore, the Coco System can auto-generate conformance checkers described below and test programs from the requirements and specifications for the other parts of the system developed using conventional methods for example, legacy code that the system under development has to interact with.

The Coco System is modelling-language and modelling-tool independent, enabling the essential interoperability between different solutions. The Coco System also provides innovative solutions for enabling automated formal verification to scale effectively in order to handle the increasing complexity and size of modern industrial software systems. Some further advantageous features of the Coco System are:

1. It uses a modelling language that can express systems of communicating components in unrestricted architectures and is a suitable target for many modelling languages. For example, it is capable of expressing models written in the open UML and SysML standards, but can also handle proprietary modelling languages. The language and the automated translators within the Coco System are the key enablers for: a) an interoperability between modelling languages and tools; b) ensuring end-to-end consistency of all views of the system, including high level models, mathematical notations for formal verification, and executable source code in different programming languages; c) enabling full traceability through the software development process.

2. Advanced fully automated formal verification technologies that provide the scalability required to verify software systems of increasing size and complexity, whilst requiring no specialised input from the user. This is provided by the development of several intermediate languages that allow the semantics of the system to be precisely specified, but also permits advanced optimisations and abstractions to be automatically performed to increase the scalability of the formal verification technology.

The system described herein yields a significant improvement in the software development process by: a) reducing the cost of software development; b) enabling more complex software systems to be developed; and c) increasing the reliability of the resulting software systems.

The Coco System has several useful features:

- It is capable of modelling not only the actual behaviour of individual components of a software system, but also the expected behaviour design intent of components. The verification capability of the Coco System can then be used to verify if a component satisfies its expected behaviour and, unlike current tools that use formal verification, also whether the system as a whole meets system-level requirements.
• It includes a novel modelling language that is capable of expressing the semantics of a variety of model-based languages such as UML and SysML. This enables interoperability between different languages and makes formal verification, code generation and test-case generation uniformly and universally available.

• It allows the resulting generated executable code to be integrated with legacy pre-existing code that is developed separately e.g. the executable code may be unverified and handwritten. The Coco System can refer to code that is written outside of the Coco System. Furthermore, the code generator within the Coco System is capable of generating test cases for the external code. It can also generate conformance checkers that are capable of verifying if external code is behaving as expected when the system is running.

• The Coco System allows the user to specify functional requirements that the system must satisfy in a variety of ways. The requirements may be specified using constructs in the Coco System. The formal verification ability of the Coco System can then be used to check if the model satisfies the requirements. In addition, the formal verification capability can be used to verify if a component of the overall model matches the behaviour expected by it.

• It includes novel formal verification technologies that allow for highly efficient scalable verification of complex software architectures. Notably, it can efficiently verify concurrent and asynchronous systems, unlike many other formal verification tools. This is enabled by including several intermediate languages that allow advanced optimisations and abstractions to be performed.

• Due to the unique combination of the Coco language (also referred to as 'Coco') and the enhanced formal verification technologies, the Coco System is able to model and verify systems with unrestricted architectures in contrast; some prior art tools only support architectures where the components are arranged in a tree.

• It is highly extensible to allow new modelling languages or alternative semantics to be easily supported via language adapters. For example, a UML language adapter could be written in such a way as to permit multiple alternative UML semantics to be supported; e.g. supporting the IBM Rhapsody state machine semantics, or the OMG standard semantics. The language adapters can also be configured by the user prior to their operation to support custom semantics, as required.

• The Coco System is also easily able to support: new executable code languages via new code adaptors added to the system; new formal verification tools via new verifiers added to the system; and automated generation of customised legacy code bindings by virtue of the legacy code adaptors in the executable code generator.

The Coco System is independent of the frontend modelling tool, thereby enabling integration with and interoperability between different modelling tools. These could either be commercial tools supplied by tool vendors (namely source-specific tools), or
proprietary tools developed in-house or open source tools. This enables advanced formal verification, code generation and test case generation to be used uniformly, universally and independently of the modelling tool environment.

The system scales for large complex industrial software systems (millions of lines of code). In particular, this requires scalable theory and technology for the formal verification aspect, which is a key point of weakness in much of the current prior art approaches.

It provides a language with a rich executable semantics including support for modelling nondeterminism that provides a single point of truth and upon which a broad range of modelling languages can be mapped. This is a key enabler for: a) interoperability between modelling languages, and modelling tools; b) access to automated formal verification backend technologies; c) semantics-preserving model verification and code generation.

The Coco System dramatically reduces the cost of software development, enables users to manage the increase in software complexity, and increases the reliability of the resulting software.

The Coco System achieves a reduction in the cost of software development by:

1. Enabling interoperability between modelling languages and model-based tools.
2. Enabling non-invasive integration into existing model-based engineering practices by integrating into existing model-based engineering tools and being accessed through existing standards-compliant modelling languages.
3. Enabling automated formal verification of individual components and the system as a whole, before code is generated, thereby detecting defects early in the software lifecycle. In turn, this leads to a reduction of rework.
4. Enabling automated code generation that preserves the semantics of the different modelling input languages and tools and is generated from verified designs.
5. Enabling increased re-use of software components between software projects due to the combination of formal verification and code generation.

The Coco System provides the users with the ability to manage the increasing complexity of software by:

1. Allowing users to formally verify the expected relationship between different abstractions of the software system.
2. Enabling interoperability between modelling language and tools, allowing the users to select different tools for different aspects of the system being developed.
3. Providing a mechanism for developing systems in a compositional way.
4. Providing scalable automated formal verification that is capable of handling large complex software architectures.
5. Allowing the user to express the system at different levels of abstraction and, furthermore, providing automated formal verification tools that allow the relationship between the different levels of abstraction to be understood.

The Coco System increases the reliability of the resulting software by:

1. Using scalable automated formal verification to exhaustively verify if the system is defect-free.
2. Using automated code generation that ensures that the code deployed is equivalent to the verified model.

Within the scope of this application it is expressly intended that the various aspects, embodiments, examples and alternatives set out in the preceding paragraphs, in the claims and/or in the following description and drawings, and in particular the individual features thereof, may be taken independently or in any combination. That is, all embodiments and/or features of any embodiment can be combined in any way and/or combination, unless such features are incompatible. The applicant reserves the right to change any originally filed claim or file any new claim accordingly, including the right to amend any originally filed claim to depend from and/or incorporate any feature of any other claim although not originally claimed in that manner.

BRIEF DESCRIPTION OF THE DRAWINGS

One or more embodiments of the invention will now be described, by way of example only, with reference to the accompanying drawings, in which:

Figure 1 is a graph illustrating the relative cost of software defects at different stages of the software development process;

Figure 2 is a block diagram which depicts a standard known model-based software development process;

Figure 3 is a block diagram showing a modified software development process in accordance with an embodiment of the present invention;

Figure 4 is a flowchart showing the general process by which formal verification is combined with automated code generation by the process shown in Figure 3;

Figure 5 is a block diagram showing the overall architecture of a system, the Coco System, in accordance with an embodiment of the present invention, comprising modelling language adapters and a Coco Platform;

Figure 6 is a flowchart showing the process by which formal verification may be combined with code generation, in the Coco System shown in Figure 5;
Figure 7 is a block diagram showing the architecture of the Coco System of Figure 5, focusing on the modelling language adaptors;

Figure 8 is a block diagram showing the architecture of the Coco Platform of Figure 5 in detail;

Figure 9 is a block diagram showing the architecture of a Coco Verifier shown in Figure 8, in greater detail;

Figure 9A is a block diagram showing the architecture of a further verifier shown in Figure 8, in greater detail;

Figure 10 is a block diagram showing the architecture of an executable code generator shown in Figure 8, in greater detail;

Figure 11 is a diagram showing an example to illustrate how SysML ports may be mapped into HL-Coco, using the modelling language adaptors shown in Figure 7;

Figure 12 is a diagram showing an example to illustrate how SysML blocks may be mapped into HL-Coco components, using the modelling language adaptors shown in Figure 7;

Figure 13 is a diagram showing an example to illustrate how SysML state machine diagrams may be mapped into HL-Coco, using the modelling language adaptors shown in Figure 7;

Figure 14 is a diagram showing an example to illustrate how SysML interactions may be represented in HL-Coco, using the modelling language adaptors shown in Figure 7;

Figure 15 is a diagram showing an example of how SysML activity diagrams may be mapped into HL Coco, using the modelling language adaptors shown in Figure 7;

Figure 16 is a diagram showing how a single HL-Coco model may be mapped into multiple LL-Coco models, using the high-level to low-level transformer shown in Figure 8 in accordance with an embodiment; and

Figure 17 is a diagram showing an example to illustrate how LL-Coco may be mapped into CSM, using the low-level-to CSM transformer shown in Figure 9 in accordance with an embodiment.

DETAILED DESCRIPTION

Embodiments of the invention are described below with reference to Figures 3 to 17.
It is to be appreciated that the term 'functional model specification' is a specification of a model describing a SUD in terms of a system having a plurality of inputs and a set of outputs generated by the system in response to the inputs. In some non-limiting but preferred embodiments the functional model specification can be described in terms of a state machine. The functional model specification can be written in any modelling language to describe the SUD.

It is to be appreciated that the term 'SUD' as used throughout this specification is a reference to all or part of a system under design (SUD), including an individual component or port itself. Commonly such systems can be complex and very large with multiple parts (or sub-systems). Accordingly, it is often the case that a portion of the system has to be modelled and verified and/or coded at a time to make the processing task manageable.

Similarly it is to be appreciated that the term 'model' as for example used in the terms 'original model' or 'transformed model' within this specification is a reference to all or part of a model, including a single component or port. Thus very large models can be handled by the present invention and embodiments because portions or parts of the model can be handled in the same manner as a complete model. This modularity is particularly useful because it enables the verification and/or code generation of the present embodiments to be readily scalable and susceptible to concurrent implementation.

Figure 3 depicts the modified software development process 300 when the Coco System 510 is used, in accordance with an embodiment of the present invention.

The software, as developed using the software development process of Figure 2, has been developed using a model-based engineering tool. However, the key difference is that any type of model can be input 325 to the Coco System 510 to be formally verified and to allow the executable code 330 to be generated automatically from the model 320, 325. If the user wishes to integrate the executable code into hand-written or legacy software 340, then the Coco System 510 automatically generates the necessary code 330 to bind to the legacy software. Further, as the Coco System 510 is capable of automatically generating tests and conformance checkers 345, these outputs can be fed into the Unit Testing 350 phase to increase the reliability of the non-Coco System components. The automatically generated code 330 and any non-Coco developed system is then integrated and tested 360. Following this the system components are combined to form the SUD and tested 370 before the final acceptance testing 380 is carried out. Requirements, design and implementation error feedback 390 is provided but only for systems developed using a non-Coco system 340, e.g. legacy software.

As discussed earlier and illustrated in Figure 1 and Figure 2, a key factor that contributes towards escalating costs and uncertainty in software development is
finding errors late during the testing phase; particularly errors relating to the requirements, system and component design phases of the development cycle. An important attribute of the Coco System 510 (See Figure 5) is that these errors are found very early by verification phase, before code is generated and before any integration testing 360 occurs. This is illustrated clearly in Figure 3 by the absence of error feedback arrows to the MBE platform after the source code has been generated by the Coco System 510.

Figure 4 is a flowchart 400 depicting the overall process by which the Coco System 510 provides its functionality. At the start of the process, the Coco System 510 receives, at Step 410, a model file and validates, at Step 420, the model. Validation of the model is carried out by model validators (709, 718, 738) as part of function of the Modelling Language Adaptors 520 (see Figure 7) and ensures that the model is well-formed before being transformed into HL-Coco. Examples of errors found during the validation process include mismatched types, undefined identifiers, and unspecified parts of the models. Model validation is described in further detail with reference to Figure 7. If the validation is successful, the Coco System 510 then verifies, at Step 430, the model, and following successful verification, executable code is generated, at Step 440. If verification is unsuccessful, the user may in certain circumstances wish for the executable code to be generated anyway, at Step 450. If, however, verification is unsuccessful and the user would like the errors to be outputted, or if validation, at Step 420, is unsuccessful, the errors are outputted and converted, at Step 460, into the modelling language in which the model is written. Optionally, the user may be given a choice as to whether or not to correct the code and resubmit the model to the Coco System 510, at Step 470. If the user chooses to correct the code and resubmit the model, the process is repeated from Step 410 in which the Coco System 510 receives the revised model file. If the user does not wish to correct the code, the process can be ended with the errors being available in the appropriate modelling language. This example shows how formal verification can be combined with automated code generation.

Figure 5 shows the overall architecture 500 of the Coco System 510, in accordance with an embodiment. The system comprises a plurality of Modelling Language Adaptors 520 and a Coco Platform 530.

The Coco System 510 inputs a model file 540, expressed in a modelling language for example, UML or SysML, which describes a system to be developed. A Modelling Language Adaptor 520 takes input in a modelling language, e.g. SysML, and translates it into High-Level Coco 550 (HL-Coco), which is a modelling language defined below. The resulting HL-Coco model 550 is input to the Coco Platform 530. The Coco Platform 530 can then either analyse the HL-Coco model 550, producing an HL-Coco analysis output 560, or the HL-Coco model can be converted into executable code 570. If an HL-Coco analysis output 560 is produced, then the Modelling Language Adaptor 520 translates the analysis output 560 into language-specific
analysis output 580 that is specific to the input MDE language e.g. the HL-Coco analysis output is converted into SysML analysis output by the SysML Adaptor.

High-Level Coco is a new modelling language that is specifically designed to allow software systems that consist of several communicating components to be expressed. HL-Coco incorporates several specific features:

- It includes a standard imperative programming language that allows computational behaviour e.g. data manipulations to be captured.
- It allows individual components to be captured, where a component can either be a single unit of behaviour, or can be a collection of other components. For each component, HL-Coco can capture:
  - The structural and behavioural relationships of a component with the other components in the collection that are required by this component to provide its functionality.
  - The actual behaviour of a component, which includes capturing the way that the component interacts with the components it requires for its functionality. These behaviours can be captured using, for example, state machines.
  - The expected behaviour of the component that describes how other components require this component to behave. Since this is often an abstraction of the actual behaviour and is therefore nondeterministic, HL-Coco has various constructs that allow the nondeterministic behaviour of a component to be expressed, for example using state machines.
  - The requirements that either the overall system, or some part of it, must satisfy. These requirements may be functional requirements or non-functional requirements.
  - The execution semantics of the individual components e.g. how does a component process incoming events.
  - It is sufficiently expressive to enable either new verification and code generation tools to be incorporated into the Coco System, or to enable existing tools to be expanded with new functionality.

In order to define how the software is organised into components, HL-Coco has several specialised constructs:

- Interface: this defines the language that can be used to interact with a component by defining the functions that are available on the Interface and are thus available to users of this component. The interface also specifies the signals that a component can produce and send to its other components i.e. these can be viewed as notifications to other components.
- Port: this is a specific interaction point on a component. A port has several fields, each of which specifies an interface that the port either provides or requires depending on its direction. In addition, a port has an associated behaviour that defines the allowed interactions over the port. This specifies the
sequences of calls and signals that are allowed over a port. This is used by a Verification Engine (described later) to verify if a component is used correctly.

- **Component**: as above, conceptually a component is some physical hardware or software element of the overall system where a component may itself comprise a collection of components. A component in HL-Coco provides ports that can be consumed by other components, and requires ports from other components. A standard component also has a description of the behaviour of the component e.g. as defined by a state machine. This description specifies later how the function calls and received signals are handled. Coco also permits a component to be external that indicates the component has no corresponding implementation. Such components are used to represent the boundary with the legacy software system.

The Coco language (also referred to as 'Coco') is also capable of expressing behaviours, for example, by using state machines. One of the novel features of HL-Coco is that it combines a powerful imperative programming language with support for expressing abstract state machines and system-level requirements. This unique combination of features is arranged in such a way as to enable efficient formal verification; something that other such languages cannot support.

As an example of some of the constructs of HL-Coco, a state machine can be defined to represent the actual or expected behaviour of an individual component within HL-Coco. A state machine has variables that store relevant data, and has one or more states. Each state specifies which functions may be called and which signals may be received in that state. Each function or signal is associated with a handler that specifies what action is to be taken in response; this is a general block of code that can manipulate variables, send signals, call other methods, etc. In order to express the expected behaviour of an individual component the Coco state machines incorporate constructs that enable nondeterminism and abstraction to be expressed. One such construct is known as offer:

```cpp
p.fn(x : Int) = offer {
    if (s == 0 && x > 0) v1;
    if (s > 0 && x == 0) v2;
    else offer {
        if (s != 0) v3;
        if (x < 0) illegal;
    }
}
```

The above construction specifies how a function call fn from the port p should be handled. The offer construction consists of several cases, where each case consists of: a Boolean guard specifying whether the case is enabled; the return value of the function when this case is selected. The term 'Boolean guard' is a check of preconditions to determine if a transition can be executed and in the example given above, the Boolean guard is the expression in parentheses and the value of the
transition is one of v1, v2 or v3. When the function call arrives, the Boolean guards are evaluated and one of the enabled cases is selected non-deterministically. If none of the cases are enabled, then the else clause is used instead. The power of the offer construct comes from the fact that they can be nested, and can contain guards; these permit a wide variety of languages to be translated into Coco with ease.

As another example of a novel feature of Coco, it is possible to express abstractions within Coco for specified types, permitting both efficient verification and runtime code generation. For example, it may be that the argument to one function has to be an unbounded integer at runtime to ensure compatibility with legacy code. However, if it is known that the integer will only take one of a small number of values, it makes sense to abstract the unbounded integer to a small bounded integer, enabling more efficient verification.

A HL-Coco file consists of several state machines that together describe the SUD.

Figure 6 is a flowchart 600 that depicts the process by which the Coco System 510 combines formal verification with code generation, in accordance with an embodiment. In particular the flowchart of Figure 6 shows the process shown in the flowchart of Figure 4 in greater detail, particularly with respect to the model verification step 430.

First, the Coco System 510 receives, at Step 410, a model file 540 and validates, at Step 420, the model. If the validation is successful, the Coco System 510 then verifies, at Step 430, the model. If, however, validation of the model is unsuccessful, the errors from the validation process may be outputted at Step 460 in an analysis output in the modelling language in which the model (original model) is written.

Following successful validation of the model at Step 420, the formal verification process begins. The dotted line in Figure 6 represents an expanded version the formal verification process of Step 430, whereby the formal verification process is described in greater detail with reference to Steps 610 to 680.

At the start of the verification process, the model is transformed, at Step 610, into HL-Coco (a high-level language called High-level Coco). The Coco System 510 also makes use of an intermediate language called Low-Level Coco (LL-Coco). LL-Coco is an extension of HL-Coco to include explicit code that fully describes the semantics of the model. For example, LL-Coco would include explicit details of the execution semantics regarding threading (for example, multi-threaded), and the processing of asynchronous signals via queues. In HL-Coco, the execution semantics of the model are implicit. In LL-Coco such details are made explicit by including additional code to actually implement the execution semantics. This split in capabilities is important and novel as it allows a single HL-Coco model to be executed in different ways without any changes.
The HL-Coco model is then transformed at Step 620 into LL-Coco. For many applications, it is extremely important that the model can be run efficiently. Thus, the LL-Coco model (also referred to as a low-level language specification) is optimised at Step 630 before any further analysis steps are undertaken. A check is carried out, at Step 640, to determine whether or not the LL-Coco model is fully optimised. If it is not, the optimisation process is repeated at Step 630. If the LL-Coco model is identified as being fully optimised at Step 640, the LL-Coco model is transformed, at Step 650, to a specialised programming language called CSM, which stands for Communicating State Machines, that is designed to be an input language to a formal verification tool.

Next, the CSM model (CSM specification) is optimised, at Step 660. At this stage, various transformations on the CSM model are carried out to ensure that the CSM model is optimised. A check is carried out, at Step 670, to determine whether or not the CSM model is fully optimised. If it is not, the optimisation of the CSM model is repeated at Step 660. If the CSM model is identified as being fully optimised at Step 670, the CSM model is verified, at Step 680, to determine whether the CSM model satisfies a given specification. Further detail about the optimisation and verification of the CSM model is provided later with reference to Figure 9.

Following successful verification, executable code 570 is automatically generated, at Step 440, and the resultant executable code 570 is guaranteed to have equivalent semantics to the original model input into the system. The Coco System 510 can also automatically generate test cases and conformance checkers to ease the integration of the executable code 570 into the overall software system.

If verification of the CSM model is unsuccessful, the user may wish for the executable code 570 to be generated anyway at Step 450. If, however, verification is unsuccessful and the user would like the errors from the verification process to be outputted, the errors are outputted in an analysis output and converted at Step 460 into the modelling language in which the original model was written. The analysis output is produced by a Verification Engine (described later) within the Coco System 510, and describes to what extent the model meets its specified requirements. If a requirement is not met, then the analysis output provides an explanation as to why not.

The steps in the process shown in Figure 6 are carried out by the components described in detail in Figures 7 to 10.

Figure 7 shows the architecture of the Coco System 510 described in Figure 5, where the system comprises a plurality of Modelling Language Adaptors 520 and the Coco Platform 530. In Figure 7, the Modelling Language Adaptors 520 shown in Figure 5 are decomposed and described in more detailed.

A Modelling Language Adaptor 520 comprises a model validator 709, 718, 738 for validating a model, a model transformer 706, 716, 735 for transforming a validated language-specific original model into the HL-Coco language (HL-Coco) and an
analysis transformer 707, 717, 740 for transforming analysis written in HL-Coco back into its specific modelling language, thus enabling the user to work fully in their existing modelling environment. The Modelling Language Adaptor 520 takes input in a modelling language (e.g. SysML), validates the inputted model to ensure that the model is well-formed and translates the validated model into HL-Coco 550. Whilst not shown, any model which fails to be validated is not passed onto the model transformer 706, 716, 735 but rather the validation errors are flagged and a notification is sent back to the location where the model was received from.

More specifically, in the embodiment shown in Figure 7, one of the Modelling Language Adaptors 520 is a UML adaptor 705 that comprises a UML model validator 709, a UML to HL-Coco transformer 706 and a HL-Coco to UML analysis transformer 707. The UML adaptor 705 takes input in UML 710 and outputs HL-Coco 550 to the platform via the UML model validator 709 and the UML model transformer 706. The UML adaptor 705 also takes HL-Coco Analysis Output 560 from the Coco Platform 530 and transforms this back into UML 708. Similarly, an SysML Adaptor 715 is provided that comprises a SysML model validator 718, a SysML to HL-Coco transformer 716 and a HL-Coco to SysML analysis transformer 717. The SysML adaptor 715 takes input in SysML 720 and outputs HL-Coco 550 via the SysML model validator 718 and the SysML model transformer 716. The SysML adaptor 715 also takes HL-Coco analysis output 560 from the Coco Platform 530 and transforms it into SysML 725.

The resulting High-Level Coco model 550 is input to the Coco Platform 530, which can either analyse the HL-Coco model, producing an HL-Coco Analysis Output 560, or the HL-Coco model can be converted into executable code 570. If HL-Coco Analysis Output 560 is produced, then the Adaptor translates the analysis output into analysis output specific to the input MDE language, e.g. HL-Coco Analysis Output is converted into SysML Analysis Output 725 by the SysML Adaptor 715.

An Adaptor 730 for any modelling language (X) comprises three components: a Model Validator 738, a Model Transformer 735 and an Analysis Transformer 740. The Model Validator 738 validates the inputted model 745. The Model Transformer 735 transforms the input language 745 into HL-Coco. The Analysis Transformer 740 transforms the analysis output 560 produced by the Coco Platform 530 which is expressed in terms of HL-Coco into analysis output 750 for the Adaptor's chosen language so that it can then be presented to the user.

Figure 8 shows the architecture of the Coco Platform 530.

The Coco Platform 530 takes HL-Coco 550 as input and has two primary functions. Firstly, it is able to verify mathematically if the model has the intended functionality i.e. if the model satisfies its specified functional requirements, using a Coco Verifier 810,
and produce an analysis output 560. Secondly, it is able to generate executable code
570 that is semantically equivalent to the HL-Coco model.

The Coco Platform 530 makes use of an intermediate language called Low-Level Coco
5 (LL-Coco). LL-Coco 825 is an extension of HL-Coco 550 to include explicit code that
fully describes the semantics (as described previously) of the model. In HL-Coco 550,
the execution semantics of the model is implicit. In LL-Coco 825 such details are made
explicit by including additional code to actually implement the execution semantics. The
split in capabilities is important and novel as it allows a single HL-Coco model 550
to be executed in different ways without any changes.

The HL-to-LL Transformer 830 takes in input in HL-Coco 550 and automatically
translates it into LL-Coco 825 by inserting additional statements to make the execution
semantics of the model explicit, according to the execution semantics that the user
wants and has specified previously in the original model itself.

As an example, the transformation in Figure 16 shows two different possible LL-Coco
representations 1610, 1615 of the same HL-Coco model 1605. This shows the power
of this approach: one HL-Coco model can be translated into a variety of different LL-
Coco models depending on which execution semantics the user wishes to use. The
user can select and configure the execution semantics they would like to assume
when creating the model, for example, via a drop-down menu in the frontend modelling
tool. This gets automatically translated into HL-Coco representation 1605 and
annotated in a verifiable manner to indicate which executable semantics should be
used.

For many applications, it is extremely important that the model can be run efficiently.
This is true not only when actually running the generated executable code 570, but also
when formally verifying if the model is correct. Therefore, using the LL-Coco
Optimiser 835, the generated Low-Level Coco 825 is optimised before any further
analysis steps are undertaken. This has the advantage that both the code generation
and the verification benefit from the same optimisations. The LL-Coco Optimiser 835
runs several optimisation passes over the Low-Level Coco 825; the order in which
these optimisation passes are run is determined by various heuristics. The Optimiser
835 chooses from many well-known optimisation techniques common to conventional
compilers, but also includes some specialised optimisation passes. For example, optimisation passes include:

- Boolean expression simplification. It is often the case that when the HL-Coco
model 550 is generated from a UML state machine that the model contains lots
of complicated Boolean expressions which are pieces of code that either return
true or false. It is therefore very important to simplify these as far as possible to
increase performance. For example, consider the Boolean expression true ||
false; this can be simplified to true, which in turn might lead to further optimisations.

- Data reduction. This optimisation pass is specific to the Coco Verifier 810 and so it is invoked when the resulting LL-Coco model 840 is being passed to the Coco Verifier 810. This optimisation attempts to remove variables whose values do not matter for the purposes of verification. For example, a model may include a variable that is required when incorporating the model into some larger external system; the value of such a variable may well not be important for the purposes of verification in which case it can be elided.

- State coalescing. It is often the case that after some of the other optimisation passes have been run, some of the states within each component are identical. Therefore, it makes sense to coalesce these states so that only one copy is required. This can lead to better performance, but also makes the resulting generated model simpler to understand.

After the LL Optimiser 835 has finished optimising the LL-Coco 825, the resulting optimised LL-Coco model 840 is passed to the Coco Verifier 810 and, in one embodiment subject to a positive outcome of the verification, to the Executable Code Generator 845, depending on what type of output the user has specified. A Control Switch 836 is provided for controlling the passing of the model in optimised LL-Coco to the Executable Code Generator 845. The Control Switch 836 is controlled by the result 841 output from the Coco Verifier 810, which in the present embodiment if positive enables the model expressed in optimised LL-Coco to be passed to the Executable Code Generator 845. The objective of the present embodiment is for the models to be verified before executable code 570 is generated; this is a key advantage of the Coco Platform 530. However, in other embodiments the Coco Platform 530 also allows executable code to be generated from models that have not been verified. Although this is sub-optimal, it may be desirable in practice; for example, in the case where there is no time available to carry out the verification of a given release of software to meet a release deadline. Also sometimes there are parts of a model which cannot be verified with an existing verification engine and so the ability to enable generation of executable code for that part of the model is advantageous. This reduces the barrier to adoption for the Coco System since it is not just limited to systems that are verifiable in their entirety: the code generation aspect of the Coco System can still be used. In this case, the Control Switch 836 can be configured to be overridden by sensing a specific user configuration in the model itself, which indicates that this model, or part of the model, does not require to be verified in order to have executable code generated. For example, the model could specify a particular type of verification engine that needs to be used to verify part or all of a model. The Control Switch 836 would, after reading this requirement, determine if that specific type of verification engine was available in the system. If not, then this would cause the Control Switch 836 to be overridden and the unverified model would be passed to the Executable Code Generator 845.
Alternative embodiments include Alternative Verifiers 811, 812 provided in parallel to the Coco Verifier 810 and also shown in Figure 8. These Alternative Verifiers 811, 812 are provided to verify mathematically if a model satisfies its specified requirements (functional or non-functional) using different types of verification engines. As with the Coco Verifier, 810 these Alternative Verifiers 811, 812 can also have an output (not shown in Figure 8) to the Control Switch 836 to enable the model, once verified by that Alternative Verifier, to be passed to the Executable Code Generator 845. An Alternative Verifier 811 is described later with reference to Figure 9A.

The components of the Coco Verifier 810 are described in detail with reference to Figure 9. As input, the Coco Verifier 810 takes the optimised LL-Coco model 840 after optimisation and attempts to mathematically verify if the model is correct. If the verification is incorrect, as output, it produces LL-Coco Analysis Output 850 which is then transformed into a HL-Coco Analysis Output 560 by a LL-to-HL Analysis Transformer 855. If the verification is correct, then as output it produces a positive result 841 which is sent to the Control Switch 836. In the alternative embodiments, the Alternative Verifiers 811, 812 also are configured similarly to feed positive verification outcomes to the Control Switch 836 and negative verification outcomes to the LL-to-HL Analysis Transformer 855.

Figure 9 shows the architecture of the Coco Verifier 810, in accordance with an embodiment. The Coco Verifier 810 is a custom verification tool that is designed specifically for verifying LL-Coco models and is responsible for performing the mathematical verification of the LL-Coco model 840. In principle, this could be any verification tool and the Coco Platform 530 is extensible in that it allows new verification tools to be added. For example, other verification tools may be supported by defining a suitable translation from LL-Coco into the tool's input language, and by defining a suitable way of translating analysis output back into LL-Coco.

The Coco Verifier 810 takes the optimised LL-Coco model 840 and attempts to mathematically verify if the model is correct. As output, it produces LL-Coco Analysis Output 850 which is then provided to the LL-to-HL Analysis Transformer 855. The Coco Verifier 810 also uses a specialised programming language called CSM, which stands for Communicating State Machines, that is designed to be an input language to a formal verification tool. CSM can express both a mathematical model of a system in addition to the properties that the model must satisfy. CSM can express both component and system-level correctness properties (requirements). In contrast to HL-Coco and LL-Coco, in CSM a system is specified at a much lower-level purely in terms of state machines. In addition, the state machines can be arranged to interact by passing messages to each other. CSM is therefore a specialised language for describing the semantics of systems that are built of communicating state machines. Note that CSM is not a suitable language for describing a system that is then
converted into executable code 570; it is instead a language suitable as input into a
formal verification tool.

The Low-Level Coco to CSM (LL-to-CSM) Transformer 915 is a component
responsible for converting LL-Coco 840 into CSM 920. Figure 17 shows an example to
illustrate how an LL-Coco representation 1705 may be mapped into an example CSM
representation 1710, in accordance with an embodiment. Both LL-Coco and CSM are
written in terms of well-known constructs which enables the mapping to be carried out.

Formal verification is an extremely computationally expensive problem and therefore it
is important that the model to be verified is as optimal as possible. The CSM Optimiser
925 is responsible for performing various transformations on the CSM model 920 to
draw that this model is optimised. The optimisations are performed in sequence until
the optimiser detects that no further optimisation opportunities are available. The
individual optimisations include:

- Abstractions: models often include more detail than is strictly necessary. For
example, a given model may use an unbounded integer where, in fact, all that
matters about the variable is whether the integer is odd or even. This
optimisation focuses on finding optimisation opportunities such as this, where
variables that have a large number of possible values can be abstracted,
yielding a more efficient model. This implements a well-studied area of
computer science; and

- Memory space usage: since formal verification often requires large amounts of
memory to verify a given property, it is extremely important to use as little
memory as possible to store individual states. Thus, one optimisation pass is
responsible for identifying variables that can be stored more efficiently. For
example, it may be the case that an integer variable can only be 0 or 1, and
thus can be stored using a single bit rather than the usual 32 bits.

The CSM Verification Engine 930 verifies whether a CSM model 920 satisfies either a
component or system level requirement previously specified in the model itself or by a
user. As input it takes a CSM model 920 and either says that the model satisfies the
property by sending a positive verification 841 to the Control Switch 836, or an
analysis output 935 explaining why the model does not satisfy the requested property.
The exact form of the analysis output 935 depends on the specification in question.
For example, if the specification is that a system is deadlock free i.e. it cannot reach a
state in which no further progress can be made, the analysis output for not satisfying
that property would consist of a path that reaches a state in which no further actions
are available. The CSM Verification Engine 930 has to overcome two main obstacles.
Firstly, it has to efficiently store huge numbers of states in a way that uses up the
minimal amount of memory. Secondly, it has to implement the formal verification
semantics that precisely specify what it means for a CSM model 920 to satisfy a given
property.

The CSM to LL-Coco Analysis Transformer 940 is a component that converts the
analysis output 935 of the Verification Engine 930, which is in terms of the CSM
model, into a Low-Level Coco Analysis Output 850, which is an analysis output in
terms of the original LL-Coco model.

As shown in Figure 8, Low-Level Coco Analysis Output 850 produced by the Coco
Verifier 810 is transformed into High-Level Coco Analysis Output 560 using the LL-to-
HL Analysis Transformer 855. This transformation process is simple, since HL-Coco
and LL-Coco are based on the same core language, and the LL-Coco model is simply
an expanded version of the HL-Coco model.

Referring now to Figure 9A, there is shown in detail an Alternative Verifier 811 which
uses SMT-based (Satisfiability Modulo Theories) verification. The general outline of
this is similar to the Coco Verifier 810 described above. The LL-Coco specification 840
is converted into a DIMACS (a standard input format for SMT-based tools)
specification 955 by the LL-to-DIMACS Transformer 950. This transformation process
is well-known to the skilled person, since the transformation process of imperative
languages to DIMACS is a well-studied area. The SMT Verification Engine 960 verifies
the DIMACS specification, producing and SMT Analysis Output 965 if there are any
problems with the verification. This output is then transformed to Low-Level Coco
Analysis Output 850 by the SMT-to-low level Analysis Transformer 970. If however the
analysis completely verifies the DIMACS specification, a signal 841 indicating a
positive outcome of the SMT Verification Engine 960 is sent to the control switch 836.
There are several known SMT Verification Engines 960 for example the Z3 Prover
(Microsoft) and the CVC4 theorem prover (from Stanford University).

In other embodiments, different types of model verifiers can be used either in parallel
to the above-described Alternative Verifier 811 or in place of it. Examples of these
different types of model verifiers are:

- UPPAAL: http://www.uppaal.org which is an integrated tool environment for
  modelling, validation and verification of real time system modelled as of timed
  automata, extended with data types (bounded integers, arrays, etc.);
- SPIN: http://spinroot.com which is an open-source verification tool used for
  verification of multi-threaded software applications; and
- LTSmin: http://fmt.cs.utwente.nl/tools/ltsmin/ which is a language-independent
  model checker that offers a wide spectrum of parallel and symbolic algorithms
to deal with state space explosion of different verification problems.

Figure 10 shows the architecture of the Executable Code Generator 845, in
accordance with an embodiment.
The Executable Code Generator 845 generates executable code 570 e.g. as C, C++ or Java according to the user's specified preferences. It can also generate executable code 570 for test cases. The Executable Code Generator 845 comprises several Executable Code Adaptors 1005, 1010, each of which handles a different output language for example; there is one Executable Code Adaptor for C and another for C++. This modular design ensures that it is easy to add support for new languages, as required. The Executable Code Generator 845 in Figure 10 shows a C++ Code Adaptor 1005 and a Java Code Adaptor 1010.

One advantage of this technology stack is that the code adaptors 1005, 1010 are relatively simple to write, given that many of the difficult transformation steps have already been performed most notably in the Coco Platform 530.

The design of the individual components of a single Executable Code Adaptor 1005, 1010 will now be described with reference to Figure 10. An Executable Code Adaptor may comprise a Core Code Generator 1015, a Test Case Generator 1020, a Conformance Checker Generator 1025 and a Legacy Code Adaptor 1030.

The Core Code Generator 1015 may generate the code that implements the LL-Coco model. Thus, it generates executable code 570 that describes both the structure of the system and also describes the behaviour e.g. as defined by a State Machine corresponding to each component in the LL-Coco model.

One important aspect of the generated code 570 is that it is able to integrate in with existing legacy code 340 by using external components. These are components implemented by hand or some other technology but not by the Coco System 510. Therefore, there can be mismatches between the expected behaviour as modelled in Coco and the actual implemented behaviour in the legacy code 340.

The Test Case Generator 1020 may generate individual test cases that check if a component has its required behaviour. The test cases are generated from the intended behaviour of the component, as it is described in the LL-Coco model in the ports on a component description. Generally, these tests are used to test if an external component is functioning correctly, but the Test Case Generator 1020 can be used to generate tests for any of the components in the system. The output of the Test Case Generator 1020 is a set of test cases in the form of executable code that can be integrated into the executable code generated by the Core Code Generator 1015.

The Conformance Checker Generator 1025 generates a module called a conformance checker that is able to verify at runtime if an external component is behaving according to its external interface, as modelled in Coco. The conformance checker is a wrapper around the external component that monitors the signals being produced and the functions being called, and checks that each call is permitted by the original state
machine, as specified in the Coco model. Various options are available for handling violations according to the user's specified behaviour, such as silently ignoring errors or logging violations to a file. The output of the Conformance Checker Generator 1025 is executable code that can be combined with the executable code generated by the Core Code Generator 1015.

The Legacy Code Adapter 1030 is a component responsible for automatically generating the bindings between the external component interfaces, and the corresponding legacy code implementations. For example, if a system includes an external component then the Core Code Generator 1015 does not generate an implementation of the external component, because the external component's implementation is provided by some existing code. The Legacy Code Adaptor 1030 is responsible for automatically generating the code that binds the actual Legacy Code implementation to the external component interface.

Legacy code is commonly specific to a given product being developed or the organisation developing it. Therefore, the Coco Platform 530 is extensible to enable customised Legacy Code Adaptors to be developed by users.

Figures 11-15 illustrates in more detail how the SysML Adaptor 715 shown in Figure 7 converts between SysML 720 and HL-Coco 550.

The purpose of the SysML Adaptor 715 is to convert from SysML 720 into HL-Coco 550 and analysis output 560 from HL-Coco into SysML 720. Translating analysis output 560 from HL-Coco 550 into SysML 720 is straightforward, and is therefore not considered further. The main SysML 720 constructs are blocks which are used to describe components, ports, state machines, activities, and interactions. All these constructs can be translated into HL-Coco 550, as illustrated in Figures 11-15, and described in further detail below.

Figure 11 shows one way in which SysML ports 1105 can be translated into Coco interfaces 1110 and ports 1115. The tables show the relevant properties on the SysML ports. The corresponding HL-Coco code 550 that would be generated by the SysML Adaptor 715 is shown on the right-hand-side of Figure 11.

Figure 12 shows one way in which SysML blocks 1205 can be translated into Coco components 1210. Here, a single SysML block X is translated into a single Coco component that refers to the corresponding Coco ports, which were translated as per Figure 11.

Figure 13 shows one way in which SysML state machines 1305 are translated into Coco state machines 1310. This translation is used when translating both the behaviour expected by a SysML port, and the implementation behaviour of a SysML block.
Figure 14 shows one way in which SysML sequence diagrams can be expressed in Coco. SysML sequence diagrams are often used to express functional requirements about a system, and by translating them into Coco, it is possible for the Coco Platform to then verify that the system satisfies the requirement.

Figure 15 shows one way in which SysML activity diagrams can be translated into Coco. SysML activity diagrams provide one way in which computations can be expressed within SysML. Translating activity diagrams into Coco is a straightforward procedure, given the existence of a formal semantics for SysML activity diagrams.

Figure 16 shows an example where a single HL-Coco model is mapped to two different LL-Coco models, depending on what execution semantics the user wishes to be used.

For example, LL-Coco includes a Lock construct that can be used to capture the runtime semantics. Locks are commonly used in software to protect a critical resource where only one component may access the locked item at once. A Lock may either be locked or unlocked and when locked, any further attempts to lock results in the caller being blocked. Locks can be used to implement many different execution semantics within LL-Coco, making LL-Coco a flexible framework for expressing component-based software systems with varying execution semantics.

Many modifications may be made to the above examples without departing from the spirit and scope of the present invention as defined in the accompanying claims.
Claims:

1. A system for generating verified software code from an original model of a system under development (SUD), the original model expressing the SUD in terms of components, structural behaviour between components and overall behaviour of the SUD, the system comprising:
   A plurality of modelling language adaptors arranged in parallel to receive the original model of the SUD and to convert the original model into a transformed model expressed in a common language, each modelling language adaptor being configured to convert an original model written in a specific modelling language into the transformed model expressed in the common language such that the system can generate verified code from the original model written in any of a plurality of different modelling languages;
   A model verifier for mathematically verifying if the transformed model satisfies at least one of its specified requirements and if the transformed model does not, generating verification analysis data expressing how the at least one specified requirements are not met; and
   A code generator for generating executable code from the transformed model once the model verifier has mathematically verified the transformed model as satisfying the at least one of its specified requirements.

2. A system as claimed in Claim 1, wherein the original model describes both an expected behaviour of the SUD and an actual behaviour of the SUD using conventional functional model specifications and each of the plurality of adaptors is arranged to generate the transformed model by converting the conventional functional model specifications into a processing language comprising computational processing commands and control function commands, the computational processing and control function commands in combination being functionally equivalent to the conventional functional model specifications.

3. A system as claimed in Claim 2, wherein the plurality of adaptors is arranged to convert the conventional model specifications into the processing language including commands which describe the structure of the SUD.

4. A system as claimed in Claim 3, wherein the plurality of adaptors are arranged to generate commands which describe the SUD as a plurality of components, namely elements of the SUD and how they operate together.

5. A system as claimed in Claim 4, wherein the transformed model is arranged to express the original model in terms of components, having port constructs and interface constructs:
   each component representing an element of the SUD or a collection of such elements and including a description of the behaviour of the component including the component's interaction with other components and one or more port constructs;
the port constructs being specific interaction points in a component; and each
port construct including an associated behaviour defining allowed interactions via the
port construct; and

the interface constructs defining the functions available on each interface of the
component to interact with another component and notification signals that a
component can produce to communicate with other components.

6. A system as claimed in Claim 5, wherein the plurality of adaptors is arranged to
generate commands which capture verifiable behaviour of a component or a port
construct in terms of the relationship between the inputs and outputs.

7. A system as claimed in any of Claims 5 to 6, wherein the plurality of adaptors is
arranged to generate commands which describe communications between
components of the SUD.

8. A system as claimed in Claim 7, wherein the plurality of adaptors is arranged to
generate commands which express the communications in the original model in terms
of signals, and/or function calls and function responses.

9. A system as claimed in any of Claims 4 to 8, wherein the expected behaviour
relates to one or more components or the system that represents a discrete part of the
SUD.

10. A system as claimed in Claim 9, wherein the model verifier is arranged to
compare the actual behaviour of the one or more components of the transformed
model with the expected behaviour of the one or more components of the transformed
model to verify the transformed model.

11. A system as claimed in Claim 10, wherein the actual behaviour of the one or
more components of the transformed model is processed in parallel to the expected
behaviour of the one or more components of the transformed model.

12. A system as claimed in Claim 10 or 11, wherein the one or more components
comprise an internal component representing a component of the SUD and an
external component representing a component outside the SUD which interfaces with
the SUD; and

the verifier is arranged to verify the equivalence of the actual and the expected
behaviour of the internal component using the expected behaviour of the external
component.

13. A system as claimed in any of Claims 2 to 12, wherein the original model
describes system requirements at multiple different levels within the SUD using
conventional functional model specifications and each of the plurality of adaptors is
arranged to generate the transformed model by converting the system requirements
described by the conventional functional model specifications into verifiable system requirements expressed in the processing language.

14. A system as claimed in Claim 13, wherein the model verifier is arranged to compare the verifiable system requirements with the behaviour of the transformed model to determine if the system requirements are met by the behaviour of the transformed model.

15. A system as claimed in any of Claims 2 to 14, wherein the plurality of adaptors is arranged to create the transformed model by simplifying the computational processing and control function commands from the conventional model specifications.

16. A system as claimed in any of Claims 1 to 15, wherein the model verifier comprises an optimiser for optimising the common language of the transformed model.

17. A system as claimed in any of Claims 1 to 16, wherein the model verifier comprises a plurality of different model verifiers arranged to work in parallel, each model verifier being arranged to verify a different aspect of the transformed model.

18. A system as claimed in any of Claims 2 to 17, wherein each of the plurality of adaptors is arranged to generate the transformed model using computational processing and control function commands which are arranged to express non-determinism of at least part of the original model.

19. A system as claimed in any of Claims 1 to 18, wherein each of the plurality of different modelling adaptors comprises a language-specific model validator arranged to receive the original model and to validate the semantics of the original model in the language of the original model and if the original model is validated, to continue processing the original model to convert the original model into the transformed model expressed in the common language.

20. A system as claimed in any of Claims 1 to 19, wherein each of the plurality of modelling language adaptors comprises a modelling language-specific analysis transformer for transforming the verification analysis data output from the model verifier and expressed in the common language, into the specific modelling language associated with the modelling language adaptor.

21. A system as claimed in any of Claims 1 to 20, wherein the common language is a high-level language and the system further comprises a high-to-low transformer for transforming the transformed model written in the high-level language output from one of the modelling language adaptors into a low-level language specification for input into the model verifier, wherein the high-to-low transformer inserts additional statements into the transformed model to make the execution semantics of the transformed model more specific.
22. A system as claimed in Claim 21, wherein the low-level language specification is iteratively optimised to remove redundancy, for example by Boolean expression simplification, removal of variables which are not part of the verification process and/or coalescing states of components, before being input into the model verifier.

23. A system as claimed in Claim 21 or 22, further comprising a low-to-high transformer for transforming the low-level language specification output from the model verifier into a transformed model expressed in a high-level language for input into the modelling language adaptor, wherein the low-to-high transformer removes additional statements previously inserted by the high-to-low transformer into the transformed model.

24. A system as claimed in any of Claims 21 to 23, wherein the model verifier comprises:
   a low-to-verification transformer for transforming the low-level language specification input into the model verifier into a model verification language specification, wherein the model verification language specification expresses the transformed model mathematically; and
   a model verification engine for carrying out a formal verification of the mathematically-expressed transformed model in the model verification language which is compatible with the model verification engine.

25. A system as claimed in Claim 24, further comprising a verification-to-low transformer for transforming the model verification analysis output from the model verification engine into a low-level language specification.

26. A system as claimed in any of Claims 1 to 25, wherein the code generator comprises a plurality of different code adaptors configured to operate in parallel, each code adaptor being arranged to generate and output a specific type of executable code, and comprising a core code generator for converting the transformed model into the specific type of executable code for that adaptor.

27. A system as claimed in Claim 26 as dependent on Claim 4, wherein the code generator further comprises a test case generator for generating individual test cases from the transformed model to determine if a component or a collection of components has its required behaviour, each component including a description of the behaviour of the component, the component's interaction with other components and one or more port constructs, and the output of the test case generator being combined with the output of the core code generator to provide the executable code.

28. A system as claimed in any of Claims 26 or 27 as dependent on Claim 4, wherein the code generator further comprises a conformance checker generator for verifying at run time if an external component with which the SUD interacts, is behaving according to its port construct, the output of the conformance generator
being combined with the output of the core code generator to provide the executable code.

29. A system as claimed in any of Claims 25 to 28 as dependent on Claim 4, wherein the code generator further comprises a legacy code adaptor for automatically generating code that binds an external component with legacy code, the output of the legacy code adaptor being combined with the output of the core code generator to provide the executable code.

30. A method of generating verified software code from an original model of a system under development (SUD), the model expressing the SUD in terms of components, structural behaviour between components and overall behaviour of the SUD, the method comprising:

- receiving the original model of the SUD at one of a plurality of modelling language adaptors and converting the original model into a transformed model expressed in a common language, the converting step comprising converting an original model written in a specific modelling language into a transformed model expressed in the common language such that verified code is generated from the original model written in any of a plurality of different modelling languages;
- mathematically verifying if the transformed model satisfies at least one of its specified requirements, and if the transformed model does not, generating verification analysis data expressing how the at least one specified requirement is not met; and
- generating executable code from the transformed model once the model verifier has mathematically verified the transformed model as satisfying the at least one of its specified requirements.

31. A system for formally verifying the correctness of an original model of a system under development (SUD), the original model expressing the SUD in terms of components, structural behaviour between components and both expected and actual behaviour of the SUD using functional model specifications, the system comprising:

A modelling language adaptor arranged to receive the original model of the SUD and to convert the original model into a transformed model expressed in a processing language such that the system can use the transformed model to verify the original model; wherein the modelling language adaptor is arranged to generate the transformed model by converting the conventional functional model specifications into the processing language, the processing language comprising computational processing commands and control function commands, the computational processing and control function commands in combination being functionally equivalent to the conventional functional model specifications; and

A model verifier for mathematically verifying if the transformed model satisfies at least one of its specified requirements and if it does not, generating verification analysis data expressing how the at least one specified requirement is not met.
32. The system of Claim 31, further comprising a code generator for generating executable code from the transformed model.

33. A system for automatically generating software code from an original model of a system under development (SUD), the original model expressing the SUD in terms of components, structural behaviour between components and both expected and actual behaviour of the SUD using functional model specifications, the system comprising:

   A modelling language adaptor arranged to receive the original model of the SUD and to convert the original model into a transformed model expressed in a processing language such that the system can use the transformed model to generate code corresponding to the original model; wherein the modelling language adaptor is arranged to generate the transformed model by converting the conventional functional model specifications into the processing language, the processing language comprising computational processing commands and control function commands, the computational processing and control function commands in combination being functionally equivalent to the conventional functional model specifications; and

   A code generator for generating executable code from the transformed model.
Figure 3

The Coco System

MBE platform B
- Requirements Analysis
- System Design
- Component Design

MBE platform A
- Requirements Analysis
- System Design
- Component Design

300

Analysis output

325

Analysis output

320

Analysis output

340

Automated test case generation

345

350

Automated source code generation

360

System Testing

370

Integration Testing

380

Acceptance Testing

390

Requirements, design and implementation error feedback
Figure 9A
Figure 10

Executable Code Generator

C++ Code Adaptor

Legacy Code Adaptor

Conformance Checker Generator

Test Case Generator

Core Code Generator

Java Code Adaptor

Legacy Code Adaptor

Conformance Checker Generator

Test Case Generator

Core Code Generator

Other Language Adaptors
If \( p \) is a UML port such that \( p \text{.type} = \text{XP} \) then

### XP

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classifier Behaviour</td>
<td>\textit{(state machine)}</td>
</tr>
<tr>
<td>Name</td>
<td>XP</td>
</tr>
<tr>
<td>Provided</td>
<td>XP_Functions</td>
</tr>
<tr>
<td>Required</td>
<td>XP_Signals</td>
</tr>
</tbody>
</table>

### XP\_Functions

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>XP_Functions</td>
</tr>
<tr>
<td>Operations</td>
<td>fn()</td>
</tr>
</tbody>
</table>

### XP\_Signals

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>XP_Signals</td>
</tr>
<tr>
<td>Signal Receptions</td>
<td>sig()</td>
</tr>
</tbody>
</table>

**Example HL-Coco Representation**

```plaintext
interface XPI {  
  function fn()  
  outgoing signal sig() 
}
```

```plaintext
port XP {  
  pin1 : XPI;  
  behaviour = machine M { ... }  
}
```

**Figure 11**
Example
HL-Coco Representation

```java
cOMPONENT X {
  PROVIDES { p1 : Xp; }
  REQUIRES { p2 : Yp; p3 : Zp; }
  REFERENCES { y : Y; }
  CONNECTIONS { p2 -> y.p4; }
  MACHINE M { }
}
```

**Figure 12**
Example HL-Coco Representation

```plaintext
machine XBehaviour {
  initialisation {
    currentState = State.S1;
  }
  state S1 {
    p1.Init() = { currentState = State.S2; }
  }
  state S2 {
    p1.UnInit() = { currentState = State.S1; }
    eager = {
      p1.S();
      currentState = State.S3;
    }
  }
  state S3 {}
}
```

Figure 13
Example HL-Coco Representation

interaction S { 
  x1 : X;
  y1 : Y;

  behaviour = {
    val r = cl.Init();
    assert(r == cl.Ok);
    expect(c2, cl, Error);
  }
}

Figure 14
Example HL-Coco Representation

function Activity1(in x : Int, in y : Int, out z : Int) : Nil {
    z = (x*x) + (y*y);
}

Figure 15
**Figure 16**

**HL-Coco Representation**

```plaintext
machine XImplementation {
  var x : Bool
  initialisation {
    currentState = State.S1;
  }
  state S1 {
    pl.Fn() = {
      x = !x;
      currentState = State.S2;
      return x;
    }
  }
  state S2 {
    pl.Sig() = {
      currentState = State.S1;
    }
  }
}
```

**Example LL-Coco Representations**

```plaintext
machine XImplementation {
  var mutex : Lock
  var x : Bool
  initialisation {
    currentState = State.S1;
  }
  state S1 {
    pl.Fn() = with mutex {
      x = !x;
      currentState = State.S2;
      return x;
    }
  }
  state S2 {
    pl.Sig() = with mutex {
      currentState = State.S1;
    }
  }
}
```

- or -

```plaintext
machine XImplementation {
  var mutex : Lock
  var functionLock : Lock
  var x : Bool
  initialisation {
    currentState = State.S1;
  }
  state S1 {
    pl.Fn() = {
      functionLock.acquire();
      with mutex {
        x = !x;
        currentState = State.S2;
        activeThread.notify();
        return x;
      }
    }
  }
  state S2 {
    pl.Sig() = with mutex {
      currentState = State.S1;
    }
  }
}
```
LL-Coco Representation

```plaintext
machine XImplementation {
    var mutex : Lock
    var x : Bool
    initialisation {
        currentState = State.S1;
    }
    state S1 {
        pl.Fn() = with mutex {
            x = !x;
            currentState = State.S2;
            return x;
        }
    }
    state S2 {
        pl.Sig() = with mutex {
            currentState = State.S1;
        }
    }
}
```

Example CSM Representation

```plaintext
machine XImplementation {
    channel pl_fn : Nil
    channel pl_fn_ret : Bool
    channel x1_mutex_lock : Nil
    channel x1_mutex_unlock : Nil

    machine XImplementation() = {
        var x : Bool;
        currentState = State.S1;
        state S1 {
            pl_fn? {
                x1_mutex_lock!();
                x = !x;
                currentState = State.S2;
                x1_mutex_unlock!();
                pl_fn_ret!x;
            }
        }
        state S2 {
            pl_sig? {
                x1_mutex_lock!();
                currentState = State.S1;
                x1_mutex_unlock!();
            }
        }
    }
```

Figure 17
INTERNATIONAL SEARCH REPORT

A. CLASSIFICATION OF SUBJECT MATTER

INV. G06F9/44 G06F11/36

ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

EPO-Internal

C. DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
</table>
| X        | Holenderski L, Poigné A: "Synchronise Workbench"  
28 April 1999 (1999-04-28) , Springer,  
ISBN: 978-3-7091-6355-9  
sections titled "Smooth integration" and  
"Formal verification" figure 1  
-----  
/\                 | 1-33                   |

Further documents are listed in the continuation of Box C. See patent family annex.

\* Special categories of cited documents:

\*A\* document defining the general state of the art which is not considered to be of particular relevance

\*E\* earlier application or patent but published on or after the international filing date

\*L\* document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified).

\*O\* document referring to oral disclosure, use, exhibition or other means

\*P\* document published prior to the international filing date but later than the priority date claimed

\*T\* later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

\*X\* document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

\*Y\* document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

\*S\* document member of the same patent family

Date of the actual completion of the international search 19 September 2017

Date of mailing of the international search report 13/10/2017

Authorized officer Del Castillo, G

European Patent Office, P.B. 5818 Patentlaan 2
NL - 2280 HV Rijswijk
Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016

Form PCT/ISA/210 (second sheet) (April 2005)
<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
</table>
