www.dspace.com

Computed Worst-Case Execution Time
fuel rate controller
Stateflow control logic implementation
Referenced model airflow calculation
Embedded M fuel calculation

Safely Excluding Serious Errors

Translation of “Schwere Fehler sicher ausschließen”
Published at: Elektronik Automotive 03/2014

Embedded Success dSPACE

In model-based development, software is developed at a high level of abstraction, i.e., as a model, and the C code implementation is automatically generated from the model. The high degree of abstraction allows intuitive application development and increases the efficiency of development. However, to verify system safety, it is not enough to regard only the model level; the properties of the C code and even of the binary machine code also have to be investigated. For example, this has to be done to determine the bounds for execution time and stack usage as required by ISO 26262 and also to prove the absence of run-time errors. Such errors can be reliably excluded by means of abstract interpretation-based static analyzers. The production code generator from dSPACE, TargetLink, has been coupled with the static analysis tools aiT, StackAnalyzer and Astrée from AbsInt to allow the analyzers – which function at implementation level – to be integrated seamlessly into the process of model-based software development. Thus, timing bugs, stack overflows and run-time errors can be detected in early development phases and reliably excluded.

In a model-based development process, the software is developed graphically at a high level of abstraction, usually by state machines and data flow charts. Thus, the executable model is also the system software specification. Code generators automatically generate the implementation from the abstract model, usually as C code. Two of the advantages are that developers can concentrate on the actual functionality without having to concern themselves with implementation details, and that model changes automatically lead to corresponding code changes. This contributes not only to a reduction in development effort, but also to an improvement in system safety.

The high degree of abstraction also has disadvantages, however: Some important system properties are no longer directly visible to the user. These include the occurrence of run-time errors in the generated C code due to erroneous model specifications, the stack consumption of the generated machine code, and the timing behavior of the machine code on the embedded target processor. This can result in serious errors not being discovered until later development phases. Violations of real-time requirements, stack overflows and run-time errors can result in incorrect system responses and even a complete system crash. Proving that these so-called nonfunctional errors cannot occur is one of the verification objectives of all current safety standards such as ISO 26262, DO-178B / DO-178C and IEC 61508.

Nonfunctional software properties such as worst-case execution time, worst-case stack usage, and the occurrence of run-time errors are hard to detect with testing and measurement methods: As a rule, specific test cases like stimulating the worst-case execution time (WCET) are not available, and there is no known safe test end criterion. If measurements are based on code instrumentation, the possibility that the result might be distorted by instrumentation has to be ruled out in the safety-critical area, and this is an unsolved problem, especially in the case of run-time measurements. The necessary test effort is usually high, and the test results are incomplete.

Formal verification methods offer a solution because they can mathematically prove the absence of errors. One such method is abstract interpretation, a formal method for static program analysis. Abstract interpretation-based static analyzers return reliable results that are valid for any possible program execution and for any possible input scenario. Because of its good scalability, it can also be used in large software projects. Today, static analyzers based on abstract interpretation that can compute the bounds for the WCET and guarantee the absence of stack overflows and run-time errors are widely used in industry and can be regarded as state-of-the-art with regard to verifying nonfunctional software properties [1].

The tool coupling of the static analyzers aiT [2], StackAnalyzer [3] and Astrée [4] with the production code generator

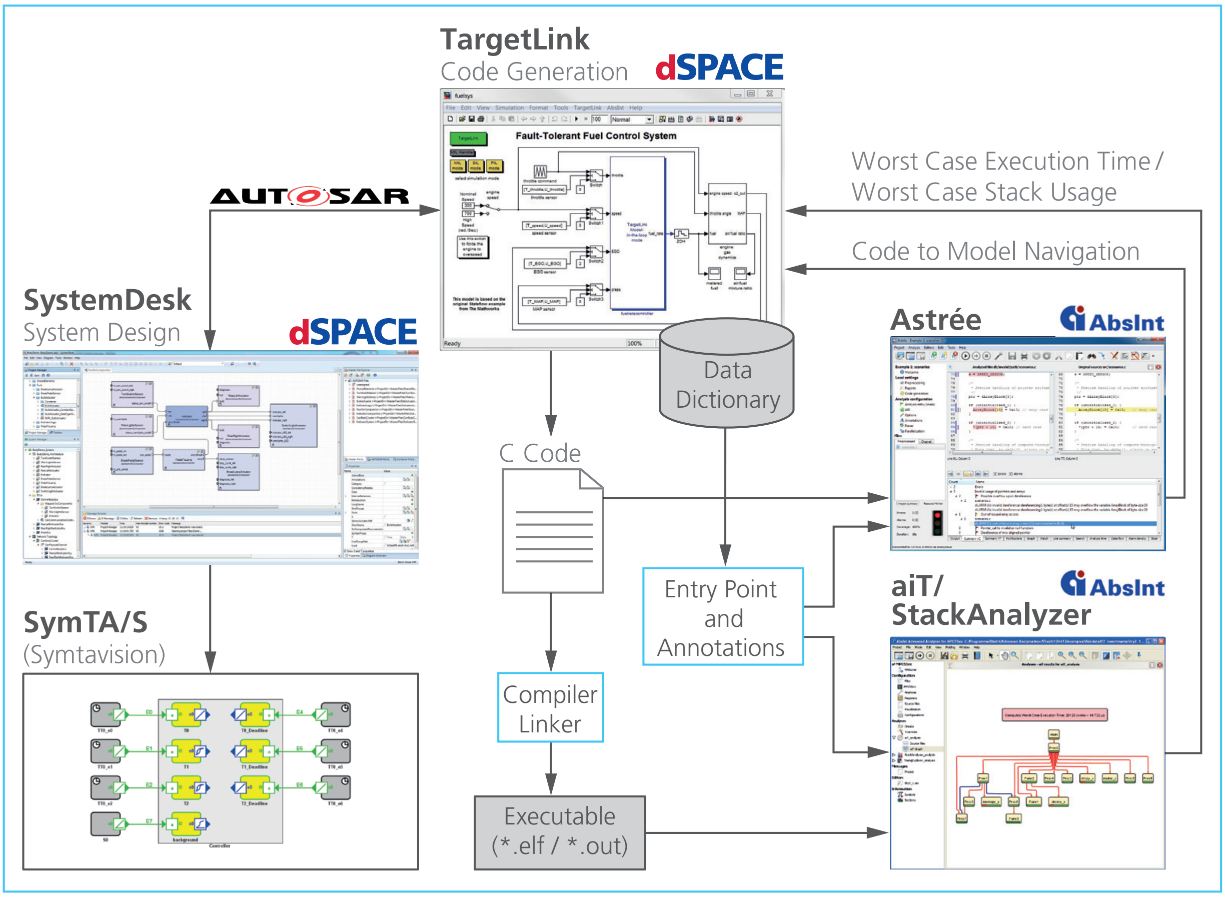


Figure 1: Workflow with the coupling of TargetLink with aiT, StackAnalyzer and Astrée.

TargetLink makes it possible to automatically calculate safe upper bounds for the WCET and maximum stack usage and also to prove the absence of run-time errors. In addition, errors are detected at an early stage in the development process so that expensive integration problems in late project phases can be avoided. Errors that are detected can be traced back to model level, and the close coupling of the tools considerably improves the efficiency of development.

Code Generation and Simulation

TargetLink® [5], the production code generator from dSPACE, generates highly-efficient C code for production-level product applications directly from MATLAB®/Simulink®/Stateflow® models. One important property of TargetLink is the separation between implementation data – on the generated functions, variables, value ranges, etc. - and the model. Because modeled software is growing in complexity, and because the associated implementation specifications have to be exchanged by different developers, a lot of data is not kept in the model itself, but in a separate, central container called the TargetLink Data Dictionary [6]. Each TargetLink model is associated to a data dictionary, and the model references the data elements that the data dictionary contains. This centralization makes it possible for different developers to work with consistent definitions of shared data, e.g., interface information or calibration data. At the same time, the TargetLink Data Dictionary is an ideal basis for connecting analysis tools.

In addition to generating the actual code, TargetLink also provides a way to execute extensive simulations at an early stage in different simulation modes. The model is interpreted in model-in-the-loop (MIL) simulation, the generated C code is executed on the host PC in software-in-the-loop (SIL) simulation, and the generated code is compiled with a cross compiler and executed on an evaluation module in processor-in-the-loop (PIL) simulation. In early design phases, these different types of simulation help check whether a model fulfills the functional requirements. Errors in the model or in other aspects such as scaling can be detected early on. PIL simulation additionally provides an easy way to perform measurements on execution time and stack usage.

As described at the beginning, the quality of these simulation- and test-based verification and validation activities strongly depends on the quality and completeness of the simulations or test cases that are performed. For this reason, reliable static analyzers should be used for safety-critical software to prove the absence of run-time errors and determine safe upper bounds for the target processor's resources.

Static Analysis of Nonfunctional Software Properties

A static analysis computes data about a software program without actually executing the program. Static analyses can be pure syntax methods such as code checkers that test for coding guidelines, *unsound* semantics-based methods, or *sound* semantics-based methods. The unsound semantics-based methods examine a program's semantics to find potential errors but cannot guarantee that all errors are found. With the sound semantics-based methods, it is possible to prove mathematically that no errors have been overlooked. They are based on the formal method of abstract interpretation [7] and over the last few years have become the state-of-the-art for verifying nonfunctional software properties [1]. The analyzers described below belong to the class of sound semantics-based methods.

Stack Usage

In embedded systems, the run-time stack is usually the only memory area that is handled dynamically. As a rule, the maximum stack usage for each task must be defined when the system is configured. If it is underestimated, stack overflows can occur. Stack overflows can cause serious errors. One example is the unintended acceleration in the 2005 model of the Toyota Camry: Testimony from expert witnesses at the US court proceedings identified stack overflow as the most probable cause [8].

StackAnalyzer is an abstract interpretation-based static analyzer that calculates bounds for tasks' maximum stack usage safely and precisely. The main input to StackAnalyzer is an executable binary file, i.e., the machine code for the target processor. The analysis requires no code instrumentation and no debug information, and precisely examines the effects of inline assembly and library functions. The analyzer calculates how the height of the run-time stack changes over the program's possible control paths and uses this to determine a safe upper bound for the maximum stack usage. The results of the analysis are visualized in a call graph and a control flow graph, and provide important clues for optimizing the stack usage.

Worst-Case Execution Time

Numerous tasks in safety-critical embedded systems have hard real-time requirements. They have to terminate within fixed time bounds to ensure that the system functions correctly. Because of the complexity of modern hardware and software architectures, determining the worst-case execution time (WCET) poses a real problem [9]. For an overview of methods and tools for WCET analysis, refer to [10].

aiT WCET Analyzer is a static analyzer that computes a safe approximation of all the target processor's possible cache and pipeline states at each point in the program. All possible program executions and all possible input scenarios are taken into account. A precise knowledge of the microprocessor architecture is necessary in order to precisely predict the number of clock cycles needed to execute the machine instructions. From this information, the longest execution path through the program can be calculated, and a safe upper bound for the WCET can be derived from that. Like StackAnalyzer, aiT works on the target processor's executable binary files. Neither code instrumentation nor debug information are needed, and the effects of inline assembly and library functions are analyzed precisely. The results of the analysis are visualized in a call graph and a control flow graph, and provide important clues for optimizing time behavior.

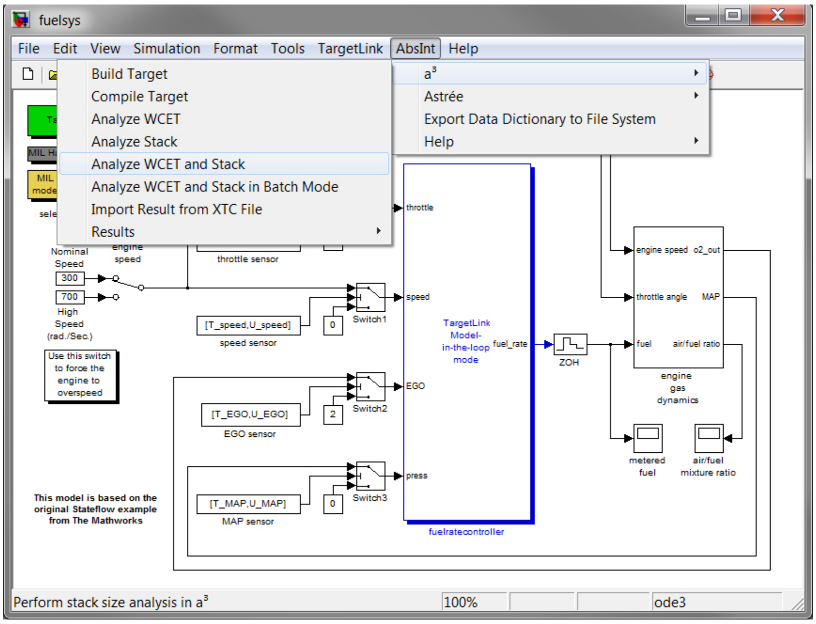
Run-Time Errors

Another class of critical programming errors is the so-called run-time errors such as arithmetic overflows, array bound violations and invalid pointer accesses. These can destroy the data integrity of a program and cause erroneous system responses or even a system crash. The explosion of the Ariane rocket in 1996 is a well-known example of the effects a run-time error can have.

One example of a static run-time error analyzer based on abstract interpretation is Astrée, which finds all the possible run-time errors in C programs and can therefore prove the absence of run-time errors. At the heart of Astrée is a highly optimized value analysis that detects relationships between variables and can precisely approximate the possible variable values. In addition, the analyzer's precision can be precisely adjusted to the software under analysis so that the available computing power is utilized efficiently. This produces very low false alarm rates at short analysis durations: Safety-critical avionics software of more than 500,000 code lines can be analyzed on an off-the-shelf PC without false alarms in 6 hours [11].

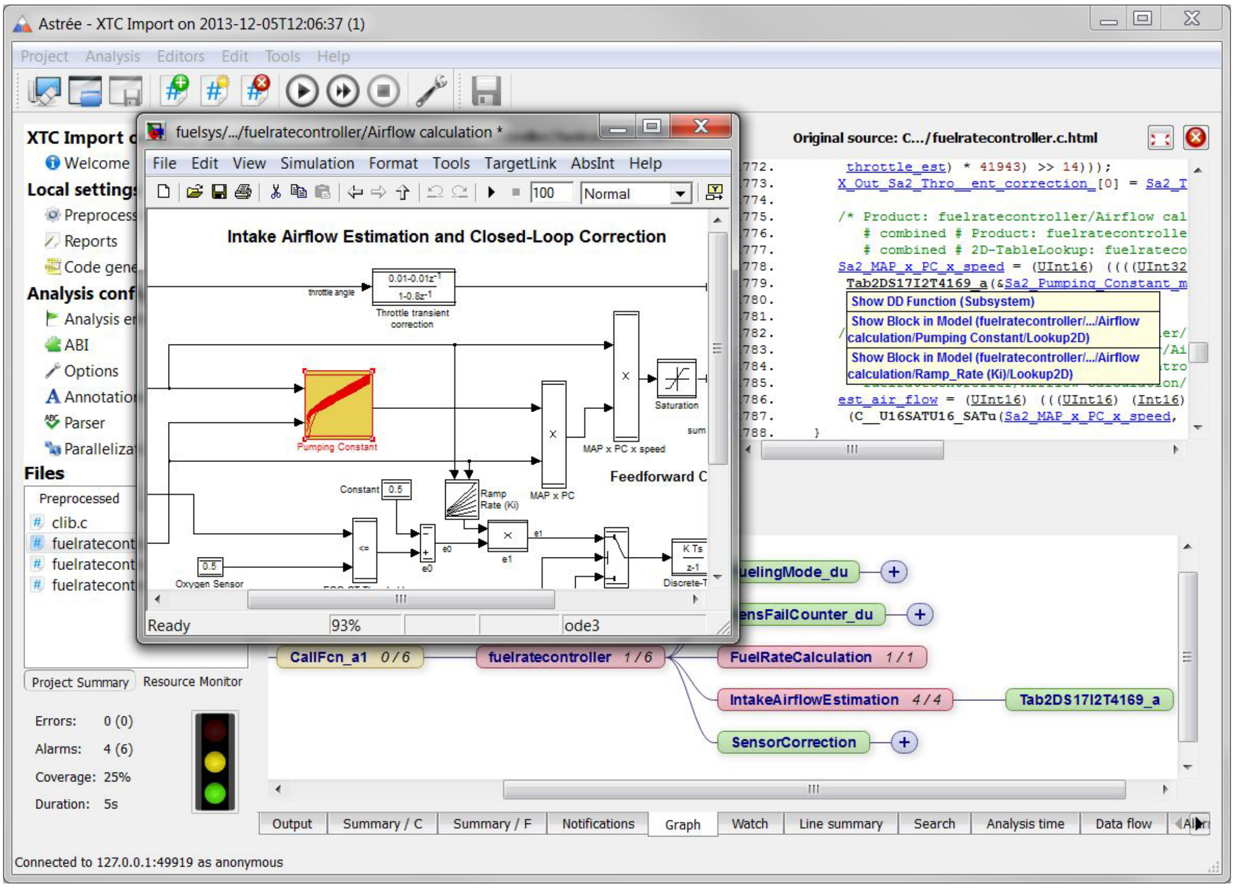
Tool Coupling

In a model-based development environment, it is easy to change and refine models and to vary the code generation options. The effects on the behavior and safety of the control system should be examined after each change.



Perform stack size analysis in a³

Figure 2: Menu of the AbsInt Toolbox.



```
    throttle_est) * 41943) >> 14)));
   X_Out_Sa2_Thro__ent_correction_[0] = Sa2_T

   /* Product: fuelratecontroller/Airflow cal
      # combined # Product: fuelratecontrolle
      # combined # 2D-TableLookup: fuelrateco
   Sa2_MAP_x_PC_x_speed = (UInt16) ((((UInt32
    Tab2DS17I2T4169_a(&Sa2_Pumping_Constant_m




   est_air_flow = (UInt16) (((UInt16) (Int16)
    (C__U16SATU16_SATu(Sa2_MAP_x_PC_x_speed,
}
```

Errors: 0 (0)
Alarms: 4 (6)
Coverage: 25%
Duration: 5s

Connected to 127.0.0.1:49919 as anonymous

Figure 3: Tracing potential run-time errors back to the TargetLink model.

Simulation-based tests are well integrated into modern model-based development environments, but reliable validation of nonfunctional requirements is usually not possible. The aim of coupling aiT, StackAnalyzer and Astrée with TargetLink is to close this gap and to make workflows within the interactive development process as efficient as possible. Moreover, it is extremely important that the analyzers can evaluate information that is contained in the TargetLink model but is not part of the generated code. By automatically including such information in the analysis, error-prone multiple specifications of the information can be avoided and the results of analysis can be considerably improved.

Figure 1 shows the workflow for calculating the WCET with the coupling between TargetLink and aiT. First the generated code is compiled and linked to produce an executable binary file. This step is not necessary for analyzing run-time errors, because Astrée works on the generated C code. Then the information on the analysis to be performed is written to an XML file in XTC format.

XTC (XML Timing Cookie) is a standardized exchange format that provides a generic data exchange interface for any desired analysis and verification tools [12]. XTC was developed as part of various international research projects, including, among others, INTERESTED, ALL-TIMES, TIMMO-2-USE and MBAT. In XTC, users can specify the type of analysis to be performed, the files to be analyzed, the entry point of the analysis, etc., and also set the analysis options. Such a project configuration is automatically generated for each root function of the TargetLink model.

The TargetLink Data Dictionary contains detailed information on the generated code: the root functions to be called, the value ranges of input and output variables, loop limits, information on interpolation functions, etc. Using this information means that a very high analysis precision can be achieved, and this is reflected among other things in a very low number of false alarms. All the relevant information is automatically converted into formal analysis directives. The annotation languages used are open formats [13] that require no modifications to the analyzed files whatsoever and that are robust toward code changes.

An entry function describing the execution model is generated for run-time error analysis. First initialization functions are executed, then the model's root functions are called from inside a reactive loop. The value ranges specified in the model are used for the input variables, or if no range specifications are available, the full value range is used. For the value ranges of output variables, static assertions are generated. These allow formal proof of compliance with the value range. The corresponding dynamic function tests can be eliminated. No particular entry point is needed for WCET and stack analyses, since the execution time and stack usage are calculated for each root function (runnable) separately.

All work steps are completely automated and can be started from the

AbsInt menu in the Simulink/TargetLink model window (see Figure 2). The analyses can also be performed automatically, for example, every time new code is generated with TargetLink or when regression tests are executed.

When the analysis completes, the WCET for each TargetLink root function (aiT), its maximum stack usage (StackAnalyzer), and messages on potential run-time errors (Astrée) are written to XML-based results files, which can be opened via the *Results* menu command in TargetLink. Information on the WCET and maximum stack usage is also saved to the TargetLink Data Dictionary. From there, it can be automatically transferred to the TargetLink model documentation or exported for AUTOSAR authoring tools such as SystemDesk in the standardized AUTOSAR format.

If TargetLink is called with the *Generate model-linked code view* option, the generated HTML files are automatically opened in Astrée. This not only simplifies the analysis of possible run-time errors, but also allows implementation errors to be traced back directly to the model level (see Figure 3).

Summary

The tool coupling described in this article provides advantages at many different levels. Program properties at implementation level – WCET, maximum stack usage, the occurrence of run-time errors – are made visible at model level. They are checked by static analyzers without the test system having to be executed on a hardware prototype. The analyzers work on the generated code and provide complete control and data coverage. The coupling can be handled efficiently and intuitively, since the static analyzers can be called directly from the TargetLink user interface – if desired, every time the model is changed. The relevant model properties are automatically converted into formal annotations of the analysis tools, which prevents multiple inputs and ensures data consistency. The connection between the analysis results and the model allows errors found at implementation level to be traced back to model level. Errors can therefore be detected at an early stage of the development process to avoid expensive integration problems in later project phases. The tool coupling specifically addresses the requirements defined in current safety standards such as ISO 26262, DO-178B / DO-178C and IEC-61508. These require proof that real-time requirements are fulfilled and that no stack overflows or run-time errors occur. The result is an automatic tool chain for developing safety-critical embedded software that combines the advantages of model-based development and static software verification.

References

[1] D. Kästner and C. Ferdinand. Efficient Verification of Non-Functional Safety Properties by Abstract Interpretation: Timing, Stack Consumption, and Absence of Runtime Errors. In Proceedings of the 29th International System Safety Conference ISSC2011, Las Vegas, 2011.

[2] AbsInt GmbH. aiT Worst-Case Execution Time Analyzer Website. http://www.absint.com/ait

[3] AbsInt GmbH. StackAnalyzer Website. http://www.absint.com/sa.

[4] AbsInt GmbH. Astrée Website. http://www.absint.com/-astree

[5] dSPACE GmbH. TargetLink Website. http://www.dSPACE.com/go/TargetLink.

[6] dSPACE GmbH. TargetLink Data Dictionary Basic Concepts Guide, November 2013.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL '77, pages 238–252, 1977.

[8] M. Dunn. Toyota's killer firmware: Bad design and its consequences. EDN Network. http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware–Bad-design-and-its-consequences, October 2013.

[9] D. Kästner, M. Pister, G. Gebhard, M. Schlickling, and C. Ferdinand. Confidence in Timing. Safecomp 2013 Workshop: Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR), September 2013.

[10] R. et al. Wilhelm. The worst-case execution-time problem — overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems, 7(3):1–53, 2008.

[11] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée: Proving the Absence of Runtime Errors. Embedded Real Time Software and Systems Congress ERTS 2, 2010.

[12] AbsInt. XTC Language Specification Version 2.1. http://www.absint.com/xtc/2013.

[13] D. Kästner, U. Kiffmeier, D. Fleischer, S. Nenova, M. Schlickling, and C. Ferdinand. Integrating Model-Based Code Generators with Static Program Analyzers. Embedded World Congress, 2013.

Dr. Daniel Kästner
is Chief Technical Officer at AbsInt Angewandte Informatik GmbH.

Carsten Rustemeier
is Product Engineer TargetLink at dSPACE GmbH.

www.dspace.com

Company Headquarters in Germany
dSPACE GmbH
Rathenaustraße 26
33102 Paderborn
Tel.: +49 5251 1638-0
Fax: +49 5251 16198-0
info@dspace.de

United Kingdom
dSPACE Ltd.
Unit B7 · Beech House
Melbourn Science Park
Melbourn
Hertfordshire · SG8 6HB
Tel.: +44 1763 269 020
Fax: +44 1763 269 021
info@dspace.co.uk

France
dSPACE SARL
7 Parc Burospace
Route de Gisy
91573 Bièvres Cedex
Tel.: +33 169 355 060
Fax: +33 169 355 061
info@dspace.fr

China
dSPACE Mechatronic Control Technology (Shanghai) Co., Ltd.
Unit 1101-1104, 11F/L
Middle Xizang Rd. 18
Harbour Ring Plaza
200001 Shanghai
Tel.: +86 21 6391 7666
Fax: +86 21 6391 7445
infochina@dspace.com

Japan
dSPACE Japan K.K.
10F Gotenyama Trust Tower
4-7-35 Kitashinagawa
Shinagawa-ku
Tokyo 140-0001
Tel.: +81 3 5798 5460
Fax: +81 3 5798 5464
info@dspace.jp

USA and Canada
dSPACE Inc.
50131 Pontiac Trail
Wixom · MI 48393-2020
Tel.: +1 248 295 4700
Fax: +1 248 295 2950
info@dspaceinc.com

03/2014

Embedded Success dSPACE