Electronic Communications of the EASST
Volume 23 (2009)

Proceedings of the Ninth International Workshop on Automated Verification of Critical Systems (AVOCS 2009)

Verification of safety requirements for program code using data abstraction

F.P.M. Stappers and M.A. Reniers

17 pages

Guest Editor: Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Verification of safety requirements for program code using data abstraction

F.P.M. Stappers[1] **and M.A. Reniers**[2]

[1] f.p.m.stappers@tue.nl [2] m.a.reniers@tue.nl
Department of Mathematics and Computer Science, TU/e,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract: Large systems in modern development consist of many concurrent processes. To prove safety properties formal modelling techniques are needed. When source code is the only available documentation for deriving the system's behaviour, it is a difficult task to create a suitable model. Implementations of a system usually describe behaviour in too much detail for a formal verification. Therefore automated methods are needed that directly abstract from the implementation, but maintain enough information for a formal system analysis.

This paper describes and illustrates a method by which systems with a high degree of parallelism can be verified. The method consists of creating an over-approximation of the behaviour by abstracting from the values of program variables. The derived model, consisting of interface calls between processes, is checked for various safety properties with the mCRL2 tool set.

Keywords: verification, safety requirements, translation, data abstraction, case study

1 Introduction

Subcontracting, buying off-the-shelf-components, and outsourcing are common in companies that develop and build embedded systems [10, 21, 30]. These companies require high quality and fault free components. Regrettably, when integrating components from different suppliers, unforeseen errors occur or unexpected behaviour is encountered [25].

Since the number of components in industrial systems grow, more lines of code are needed to control the system's behaviour [27]. To ensure that shipped systems are fault-free, tests are performed. Unfortunately, the absence of errors cannot be guaranteed by executing tests.

To prove the correctness of a system formal methods like model checking are needed [7]. Usually, these models are built from the available documentation. However, if a system is developed under pressure (e.g., prototyping, limited resources, etc.) or hardly any information is available, the implementation often becomes the main source for deriving behavioural models.

Deriving models from documentation is hard. Creating usable models from source code is even harder. Without any abstraction techniques, the models are too big to be used for the analysis of behavioural properties. We observe that many relevant properties can be stated in terms of the interface calls between processes. By abstracting from internal actions, it is possible to combat state space explosions. Nevertheless, the resulting models are still too large, because conditions (depending on values of program variables) determine if interface calls take place.

In this approach we abstract from variables and assignments and therefore systematically explore every alternative for every condition. This way an over-approximation of the possible interactions between the components is created, preserving a simulation relation [29]. The approach can be used for verifying safety requirements [19] on the interface communication in the sense that any safety requirement that holds for the over-approximation also holds for the real system.

The goal of this paper is to assess the feasibility of the method sketched above by means of a case study. Since modern languages consist of many features and hierarchical structures, this paper assumes that the source code for the control software of embedded systems is written in a simplified concurrency programming language (SCPL, Section 3). SCPL incorporates the core features for describing concurrent imperative programs and the constructs found in the studied application. Note that we do not incorporate object oriented design and complex data structures issues such as classes, inheritance and templates. We conjecture that most contemporary programming languages can be translated to mCRL2, though this needs further investigation. We also argue that it is possible to extend SCPL, such that it explicitly deals with communication. In order to prove the practical value of the method it has been executed by hand on a large case study consisting of 236 parallel threads. Based on our case study, we see no problems to automate the method.

Programs written in SCPL are transformed to models in mCRL2 [14] for which safety requirements are verified. To show feasibility, the method is demonstrated on the implementation of a controller for a printer that manufactures Printed Circuit Boards (PCBs).

This paper is structured as follows. Section 2 gives a brief introduction to the relevant fragments of the language mCRL2 and the modal μ-calculus. Section 3 describes the translation from SCPL to mCRL2 used to acquire the model for the different components. In Section 4 and Section 5 the abstraction technique is applied on an industrial system. It describes the system and the framework on which the case study is demonstrated. The case study demonstrates that we were able to prove useful requirements for this complex system. Section 6 discusses related work. Section 7 concludes with our results, discussion and future work.

2 Preliminaries

2.1 Syntax and semantics of mCRL2

An mCRL2 process is built from data-parameterised multi-actions and a collection of process operators. In this paper, a fragment of the syntax of the un-timed mCRL2 language is used. It is given by the following *BNF*:

$$\begin{array}{lcl} P & ::= & \alpha \mid P+P \mid P\cdot P \mid P \,\|\, P \mid \partial_B(P) \mid \Gamma_V(P) \mid X \\ \alpha & ::= & \tau \mid \mathsf{a}(\vec{d}) \mid \alpha \mid \alpha \end{array}$$

The small $\mid$ indicates a choice between symbols in the expression of the BNF. In this syntax α denotes a multi-action. A multi-action consists of actions combined by the big $|$. The empty multi-action is denoted by τ. An action $\mathsf{a}(\vec{d})$ consists of an action name a and possibility a data parameter vector $\vec{d}$ (the syntax of which is left unspecified). A multi-action represents the simultaneous execution of the constituent actions.

Processes are denoted by P. For processes, $+$ denotes non-deterministic choice, i.e., a choice between behaviours, $\cdot$ denotes sequential composition, i.e., a process followed by another process, and $\|$ denotes parallel composition, i.e., the interleaved execution of both processes. The operator ∂_B blocks all actions from set B of action names, i.e., prevents the occurrence of the specified actions. Γ_V applies the communications described by the set V to a process. A communication in the set V is of the form $\mathsf{a}_1 \mid \cdots \mid \mathsf{a}_n \rightarrow \mathsf{a}$. Application of Γ_V to a process means that any occurrence of the multi-action $\mathsf{a}_1(\vec{d}) \mid \cdots \mid \mathsf{a}_n(\vec{d})$ is replaced by $\mathsf{a}(\vec{d})$, for any $\vec{d}$. X is a reference to a process definition of the form $X = P$, i.e., the process X behaves as prescribed by P.

The semantics associated with an mCRL2 process, as used in the mCRL2 tool set, is a transition system where the transitions are labelled by multi-actions. A more elaborate description of the syntax and (timed) semantics are given in [13, 14].

2.2 Modal μ-calculus

Modal μ-calculus formulae are used to describe behavioural properties. These properties are then automatically verified against a behavioural model described in mCRL2. Modal formulae are specified in a variant of the modal μ-calculus extended with regular expressions [12] and data. The restricted fragment of the modal μ-calculus used in this paper is as follows:

$$\begin{array}{lll} \phi ::= & \mathit{false} \mid [\rho]\phi & \\ \rho ::= & a \mid \rho \cdot \rho \mid \rho^* & \\ a ::= & \mathsf{a}(\vec{d}) \mid \neg\mathsf{a}(\vec{d}) \mid \mathit{true} & \end{array}$$

In this syntax, ϕ represents a property, ρ represents a set of sequences of actions and a represents the presence of a data parameterised action $\mathsf{a}(\vec{d})$, the absence of a data parameterised action $\neg\mathsf{a}(\vec{d})$, or any given action (represented as *true*). The property *false* holds for no model. The property $[\rho]\phi$ states the property that ϕ holds in all states that can be reached by a sequence described by ρ. To describe action sequences concatenation and iteration can be used. A more elaborate description of the μ-calculus and its semantics can be found in [5, 12].

3 Modelling the systems behaviour

Creating a model that preserves the essentials of a system, which is still useful for simulation or verification purposes is difficult. Depending on the requirements that are to be verified, different approaches and abstraction techniques need to be used. For large systems, the abstraction needs to be chosen such, that these properties can still be verified. In this paper, we try to verify safety requirements for a system, for which the behaviour is specified in more than 200 concurrent processes. If all statements are translated without a proper abstraction, it is merely impossible to verify properties due to the well-known state space explosion problem. Therefore a systematic method is required that transforms code into a useful model, appropriate for current model checking techniques [7]. Because the requirements can be formulated in terms of interface calls between concurrent processes, the abstraction is performed on the internal operations of the individual processes. By abstracting from internal data (e.g., values of variables), conditions cannot

be evaluated accurately. Therefore, conditionals are replaced by non-deterministic choices between the alternatives. This creates an over-approximation of the systems behaviour, because potentially more behaviour can happen. If a safety property holds for the over-approximation, it must hold for the real system. On the other hand, if a safety property does not hold for the over-approximation, it may still hold for the real system.

As indicated in the Introduction, we describe our approach in *Simplified Concurrency Programming Language* (*SCPL*). With SCPL it is possible to specify a parallel program, because it has a notion of concurrency. The syntax of SCPL is described by the following BNF:

$$
\begin{array}{lll}
\langle \texttt{program} \rangle ::= & \langle \texttt{program} \rangle \, \langle \texttt{process} \rangle \;\mid\; \langle \texttt{process} \rangle \\
\langle \texttt{process} \rangle ::= & \textbf{proc}\ C = \langle \texttt{statement} \rangle\ \textbf{return} \\
\langle \texttt{statement} \rangle ::= & \textbf{call}\ N \;\mid\; \mathbf{x} := \mathbf{e} \;\mid\; \langle \texttt{statement} \rangle ; \langle \texttt{statement} \rangle \;\mid\; \\
& \textbf{if}\ b\ \textbf{then}\ \langle \texttt{statement} \rangle\ \textbf{else}\ \langle \texttt{statement} \rangle\ \textbf{fi} \;\mid\; \\
& \textbf{while}\ b\ \textbf{do}\ \langle \texttt{statement} \rangle\ \textbf{od} \;\mid\; \textbf{do}\ \langle \texttt{statement} \rangle\ \textbf{od} \;\mid\; \\
& \textbf{suspend} \;\mid\; \textbf{resume}\ N
\end{array}
$$

A program consists of at least one process. A process consists of a unique identifier, the process identifier, and a body: a process with process identifier C and body of statements S is specified by means of **proc** $C = S$ **return**. It is assumed that each program contains a process with process identifier *init* that represents the process that is to be activated initially. The body of a process consists of statements that denote calls to other processes **call** N (where N is a non-empty set of process identifiers), multi-assignments $\mathbf{x} := \mathbf{e}$, sequential compositions $S;S'$, conditionals **if** b **then** S **else** S' **fi**, the (in)finite repetitions **while** b **do** S **od** and **do** S **od**; and statements **suspend** and **resume** N for the suspension and the continuation of (sets of) processes.

We do not present a formal semantics of this language, because these programming constructs are relatively well-known. An informal semantics is given in the upcoming sections.

3.1 Translation Scheme

The translation function $\mathscr{A}$ takes a program written in SCPL and produces an mCRL2 specification, i.e., a tuple consisting of an initial process in mCRL2 and a set of mCRL2 process equations. For each process with identifier C in the SCPL program, there exists a process equation defining the recursion variable X_C in mCRL2. In the translation the following actions are used

- $Start_s(C)$ denotes a request for starting process C;
- $Start_r(C)$ denotes acceptance of the request for starting process C (by process C);
- $Done_s(C)$ denotes the return of process C (by C);
- $Done_r(C)$ denotes notification of termination for a run of process C;
- $Suspend_s(C)$ denotes the suspension of process C. If a process gets suspended the calling process interprets the suspend signal as the relevant part of the process is finished and the calling process can continue;

- $Resume_r(C)$ denotes the acceptance of the request to resume process C;
- $Resume_s(C)$ denotes the request to resume process C that is suspended.

Start, *Done*, *Suspend* and *Resume* denote the synchronizing actions between corresponding requests, which will be explained later in this section.

Assuming that the name of the initial procedure is *init*, the translation function $\mathscr{A}$ is defined as:

$$\mathscr{A}(p_1 \cdots p_k) = (\partial_{Bl}(\Gamma_E(\Gamma_B(Start_s(init) \cdot Done_r(init) \parallel (\Big\|_{C \in P_D} X_C)))), \bigcup_{i=1}^{k} \mathscr{A}'_{\chi_i}(p_i))$$

where

- $Bl = \{Start_s, Start_r, Done_s, Done_r, Resume_s, Resume_r, Suspend_s\}$ denotes the set of blocked actions;
- $B = \{Start_s \mid Start_r \rightarrow Start,\ Done_s \mid Done_r \rightarrow Done\}$ denotes primitive communications;
- $E = \{Suspend_s \mid Done_r \rightarrow Suspend,\ Resume_s \mid Resume_r \rightarrow Resume\}$ denotes the set of additional communications;
- $\Big\|_{j \in J} X_j$ describes the processes running in parallel and is recursively defined as:

$$\Big\|_{j \in \emptyset} X_j = \tau, \qquad \Big\|_{j \in J \cup \{k\}} X_j = X_k \parallel \Big(\Big\|_{j \in J \setminus \{k\}} X_j\Big);$$

- the sets χ_i of process identifiers are pairwise disjoint and are disjoint from the set of recursion variables used to capture the processes defined within the program;
- $\mathscr{A}'$ denotes the translation function for processes which is defined in the rest of this section.

The encapsulation operator ∂_{Bl} and communication operators Γ_E and Γ_B are applied to the parallel composition of the processes to synchronize successful interface calls between processes and to block individual non-successful interface calls. The different local communication operators Γ_E and Γ_B are required to guarantee unique solutions. For example, $\Gamma_{\{a|b \rightarrow c, a|d \rightarrow e\}}(a|b|d)$ has multiple outcomes, namely $c|d$ and $e|b$.

Each process of the program is associated with at least one mCRL2 process equation by means of the translation function $\mathscr{A}'_{\chi_i}$: one of these corresponds to the translated process, while the others are introduced to capture repetitions in the body of a process. To ensure that the introduced recursion variables differ from other recursion variables, the translation function is parameterized by a set of recursion variables χ_i that are free to be used and are chosen sufficiently large.

We assume that the initialization process *init* can only be called from outside the system.

3.2 Processes

Processes decompose the system's functionality into smaller manageable parts, where each process carries out a specific task. If a task is too complex for a single process it is often refined by invoking other more basic processes. The behaviour of a process can be implemented as a

function, subroutine, procedure or some functional behaviour. The behaviour of an individual process is defined by statements placed in some order.

Let **proc** $C\ = S$ **return** denote the implementation of a process, where C defines the process identifier and S defines the control flow and data transformations. A process can be invoked by using a call and when the process completes the set of tasks it will notify the calling process with a return.

In SCPL all processes that can be addressed are defined in the program. A process is either busy (by performing tasks) or idle. A busy process can become temporarily idle, until another process addresses the suspended process to continue. For processes that not have been suspended (e.g., are idle), the resume will not activate the execution of a process (e.g., the process stays idle). The translation function for a process is denoted by $\mathscr{A}'_{\chi}$, where χ denotes the set of available recursion variables. The translation function for process identifier C and statement S is given by:

$$\mathscr{A}'_{\chi}(\textbf{proc}\ C\ = S\ \textbf{return}) = \left\{ \begin{array}{rl} X_C = & Start_r(C) \cdot t_p \cdot Done_s(C) \cdot X_C \\ + & Resume_r(C) \cdot Done_s(C) \cdot X_C \end{array} \right\} \cup E_p$$

where $(t_p, E_p) = \mathscr{A}''_{\chi,C}(s)$ and $\mathscr{A}''_{\chi,C}$ is the translation function for statements as defined in the following subsection. The first summand of the equation specifies the starting of the process. The second summand is used to reflect the call for resuming an idle process. The translation function is parameterised by the identifier C of the process that is being translated. This identifier is later used to notify that a process is suspended.

3.3 Statements

In this subsection the transformation $\mathscr{A}''_{\chi,C}$ of statements is discussed. Let p and q denote statements and b a Boolean expression.

Interface calls An interface call contains a non-empty set of process identifiers. If the set contains one element, it behaves as a call to a single process. If the set contains more elements, it behaves like a call to multiple process, which need to be executed concurrently. A call simultaneously enables the start of the processes referred to in the set N. Processes can only be started if they are idle. If a call is addressed to a busy process, the call is postponed until the process becomes idle after completing the current task entirely. For processes that are temporarily idle the call is also postponed. A process that performs a call, resumes after all called processes have either suspended or completed their tasks. The interface call statement is translated as follows:

$$\mathscr{A}''_{\chi,C}(\textbf{call}\ N) = \left(\Big|_{n \in N} Start_s(n) \cdot \Big|_{n \in N} Done_r(n), \varnothing \right)$$

where $\Big|_{n \in N}\ \alpha(n)$ is inductively defined as:

$$\Big|_{n \in \varnothing}\ \alpha(n) = \tau, \qquad \Big|_{n \in N \cup \{k\}}\ \alpha(n) = \alpha(k) \mid \Big|_{n \in N \setminus \{k\}}\ \alpha(n).$$

Since there is no need to introduce additional process equations, the second element is empty.

Assignments The multi-assignment statement $\mathbf{x} := \mathbf{e}$ defines the atomic value update for the variables $x_1, \ldots, x_n$ with the values of $e_1, \ldots, e_n$. As discussed earlier, we choose to abstract from variables and the assignments to those. A multi-assignment is translated as follows:

$$\mathscr{A}''_{\chi,C}(\mathbf{x} := \mathbf{e}) = (\tau, \varnothing)$$

where the assignment itself is translated to an internal non-observable action; there is no need for additional process equations.

Sequential composition Almost every imperative programming language allows the execution of statements in a sequential order. It is evident, that the control flow depends on the sequential order and needs to be preserved. The translation for the sequential composition is as follows:

$$\mathscr{A}''_{\chi,C}(p \ ; \ q) = (\mathscr{A}''_{\phi,C}(p) \cdot \mathscr{A}''_{\psi,C}(q), E_p \cup E_q)$$

where ϕ and ψ are sets of recursion variables such that $\phi \cap \psi = \emptyset$ and $\phi \cup \psi \subseteq \chi$. These sets can always be chosen large enough to allow for the subsequent translations to have enough fresh recursion variables available. Take for example the set of recursion variables that contains a unique variable for each loop.

Conditionals The evaluation of a conditional depends on the values of variables. By abstracting from the values of variables, it is impossible to determine the outcome of a condition. Therefore, conditionals are modelled as non-deterministic choices. The conditional statement is translated as follows:

$$\mathscr{A}''_{\chi,C}(\textbf{if } b \textbf{ then } p \textbf{ else } q \textbf{ fi}) = (\mathscr{A}''_{\phi,C}(p) + \mathscr{A}''_{\psi,C}(q), E_p \cup E_q)$$

where ϕ and ψ are sets of recursion variables such that $\phi \cap \psi = \emptyset$ and $\phi \cup \psi \subseteq \chi$. These sets can always be chosen large enough to allow for the subsequent translations to have enough fresh recursion variables available.

Loops Loops are used to repeat statements that need to be carried out several times in succession. Loops are either used for computational purposes (for which they need to be finite) or for controlling the control flow (possibly infinite).

Loops are modelled by means of recursion variables. If a control loop is finite, it has a condition which determines whether or not to abort the loop. Such a conditional choice is modelled as a non-deterministic choice (as is the case for conditionals). Of course, for infinite loops, there is no reason to introduce such non-determinism.

The reason for having infinite loops in SCPL is that virtually all systems have a part that needs to run continuously during executing and for which it is not possible to abort this process. In these circumstances it must not be possible to end the control flow. An infinite loop and a finite loop are translated as follows:

$$\begin{aligned} \mathscr{A}''_{\chi,C}(\textbf{do } p \textbf{ od}) &= (Y, \{ Y = t_p \cdot Y \} \cup E_p) \\ \mathscr{A}''_{\chi,C}(\textbf{while } b \textbf{ do } p \textbf{ od}) &= (Y, \{ Y = t_p \cdot Y + \tau \} \cup E_p) \end{aligned}$$

where Y denotes a fresh recursion variable from χ, and t_p and E_p are defined as $(t_p, E_p) = \mathscr{A}''_{\chi\setminus\{Y\},C}(p)$. Note that an additional τ is required for a finite loop to make a non-deterministic choice between loop continuation or loop termination.

Processes suspension If a system runs multiple parallel processes, it is often desired to suspend the execution of a process before it may proceed. For this reason SCPL has a statement that suspends a process.

$$\mathscr{A}''_{\chi,C}(\textbf{suspend}) = (Suspend_s(C) \cdot Resume_r(C), \varnothing).$$

Process continuation Processes that are suspended become (temporarily) idle, until another process requires the continuation. SCPL offers a solution, that enables the continuation of a suspended processes by means of a resume statement. Note, that when a process is idle (after completing a task), it cannot be resumed or started by a resume. As for interface calls, multiple processes can be resumed (concurrently) by a single resume. The resume statement is translated as follows:

$$\mathscr{A}''_{\chi,C}(\textbf{resume } N) = \left(\Big|_{n\in N} Resume_s(n) \cdot \Big|_{n\in N} Done_r(n), \varnothing \right).$$

To illustrate the translation, consider the following example.

Example 1 (Translation by example) *Consider a system with two concurrent processes init and P. Process P has two computational parts, that should be suspended in between. init calls P and waits until P finishes the first computational part. When finished, the init process resumes P in order to execute the second part.*

proc *init*	=	**call** *P*;	**proc** *P*	=	*b*:= *true*;
		resume *P*			**suspend**;
		return			*b* := *false*
					return

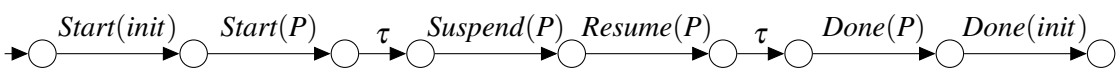
After applying the transformation we obtain the following mCRL2 specification:

$$\begin{aligned} X_{init} &= Start_r(init) \cdot Start_s(P) \cdot Done_r(P) \cdot Resume_s(P) \cdot Done_r(P) \cdot Done_s(init) \cdot X_{init} \\ &+ Resume_r(init) \cdot Done_s(init) \cdot X_{init} \\ X_P &= Start_r(P) \cdot \tau \cdot Suspend_s(P) \cdot Resume_r(P) \cdot \tau \cdot Done_s(P) \cdot X_P \\ &+ Resume_r(P) \cdot Done_s(P) \cdot X_P \end{aligned}$$

with the following initialization:

$$\partial_{Bl}(\Gamma_E(\Gamma_B(Start_s(init) \cdot Done_r(init) \parallel X_{init} \parallel X_P))).$$

The corresponding labelled transition system is depicted below:



4 Industrial application

To test the approach, it is applied on an industrial system, called "Lunaris" [26]. The Lunaris is an Etch Resist Printer, intended to operate in the manufacturing of printed circuit boards (PCBs). In current PCB production processes, the substrate is laminated with a photo resist and using a lithographic process the desired photo mask is created on the substrate. With the development of the Lunaris, it is possible to skip the expensive task of creating the mask that is required for illuminating the photo resist. By directly printing the resist in the desired pattern, it is possible to create customized and individual PCBs at lower costs.

This prototype printer has been developed for one year and has been extensively tested within this period. While the system has many physical components, we limit ourselves to verify behavioural system requirements at the level of the controller. At controller level, the Lunaris consists of 245 multi-threaded tasks (running in parallel) that are implemented in C# [1]. The tasks specify behaviour for amongst others printing, movement of physical components, logging and error handling. In total 170.000 lines of code are needed to implement the behaviour. The code is distributed over 120 classes in 40 files.

Translating the code to mCRL2 directly is possible, however this will make any exhaustive verification technique useless. A brief analysis shows that more than 10^{1000} transitions are needed, if we want to incorporate all behaviour. For this reason we apply the abstraction techniques proposed in Section 3.

The Lunaris has 7 different axes over which mechanical components move. The areas in which they operate overlap each other. If two such mechanical components operate in the same area they may collide and cause damage to the system. By means of special rules in the controller this should be prevented.

The controller must be defined by using a predefined set of tasks. In turn, these tasks can execute other tasks, which are not directly available to the controller. Since we do not have access to the implementation of the controller, we allow the predefined tasks to be executed in arbitrary order, when performing the verification.

Different tasks run in parallel, but it is not possible to run the same task simultaneously multiple times. Every task belongs to a certain activity type, e.g., logging, error handling, time delaying, ignore errors, operate hardware, etc.

The tasks are called via a master-slave protocol. A task is a master if the task itself requests execution of another task. A task is a slave if another task requests its execution. We assume that the communication takes place over non-lossy channels. The following message types are communicated between tasks:

Start A Master wants to start a task on a slave.

Done A Slave indicates that a task has been successfully terminated.

Resume A Master wants to resume a task on a slave.

Suspend A Slave suspends the current process and notifies the master.

By means of several simplifications we obtain processes described in SCPL from the C# code for tasks. First, we only consider the tasks that are relevant for manufacturing a product. Therefore, nodes that are exclusively used for logging and error-handling local to components are

excluded. This can be decided based on the activity types of the tasks. Second, we only consider "good weather" behaviour. "Good weather" behaviour is the assumption that the components behave without faults. This means that a printhead is not broken, the system prints when it is supposed to, communication channels are not lossy, etc. Third, we assume that protocols used for communication are handled correctly by the framework and the embedded software is implemented according to the specification. For this reason we do not have to specify and verify the software that provides the communication or the software on the embedded systems. Fourth, the execution of a task requires a certain amount of time. We decide not to model time aspects. This way, we prove that the correctness of the controller is not affected by performance. Note that this decisions prohibits us to verify performance properties. Fifth, for the initialization we assume the system is turned off and all components are positioned such that they reside in their initial position. Finally, we apply a pre-processing step to the classes. We know for every class the number of objects it produces. Therefore we can transform the object oriented program into a procedural multi-threaded program.

After these simplifications, we obtain 236 single threaded process, running in parallel. Based on their type of behaviour, the task templates can be decomposed into "execute tasks" and "switch tasks."

4.1 Execute Tasks

An execute task is a task that is executed once. An example of an execute task is moving the printhead device to a given position. When started, an execute task automatically completes after a finite amount of processing time.

The semantic structure of an execute task becomes as the hierarchical state machine depicted in Figure 1. A sub-task is indicated by a rectangle. Single lined boxes indicate that the sub task consists of a single state. Double lined boxes indicate that the sub task is a hierarchical state machine, which can send and receive different types of calls.

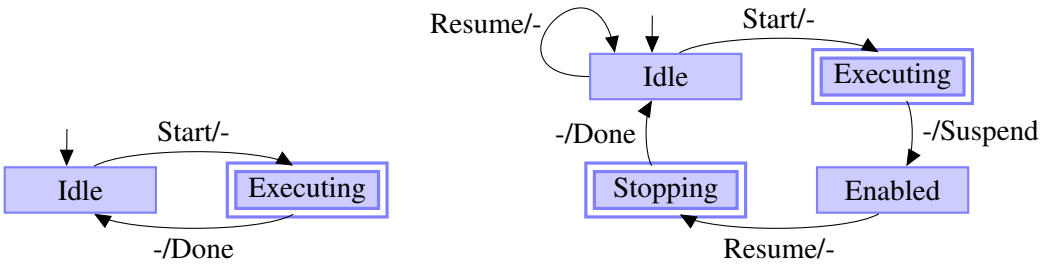


Figure 1: State diagrams of an Execute Task and Switch Task

The behaviour of an execute task with identifier *C* can be mapped to a process of the form:

proc *C* = "Executing" **return**

4.2 Switch Task

A switch task is a task that whenever started, needs to be stopped explicitly. Switch tasks are often used to enable hardware components (e.g., to enable controllers if the system reaches a certain run-level).

If a switch task is invoked, it first executes some behaviour, after which it comes into a stable enabled state. There it waits, until it receives an external signal to resume and finalize the task. Tasks that call a switch task, continue after the called switch task reaches the stable enabled state.

A switch task can be mapped to the hierarchical state machine depicted in the right of Figure 1. The behaviour of a switch task with identifier C is mapped to a process of the form:

proc C = "Executing"; **suspend**; "Stopping" **return**

5 Verification of framework properties

To validate that the technique can be used to verify safety requirements, this section discusses the requirements that have been verified with the help of the obtained mCRL2 specification. The actions used in the formulae are obtained from the mCRL2 specification.

To ensure correct behaviour, safety rules are formulated for the architecture. A safety rule is a condition that may not be violated by the execution of an action. The Lunaris has two kinds of safety rules. The first set consists of eight rules which represents warnings. If such a rule is violated, the system will raise a warning, but will continue to operate. These safety requirements are of the form: The "Switch Task (ST)" must be running if the "Execute Task (ET)" is executed. Such properties are expressed by the following formulae template:

- An execute task ET may not be started before the switch task ST is "Enabled:"

$$\begin{array}{ll} & [(\neg Suspend(ST))^* \cdot Start(ET)]false \\ \wedge & [true^* \cdot Start(ST) \cdot (\neg Suspend(ST))^* \cdot Start(ET)]false \end{array}$$

- An execute task ET may not be stopped after the switch task ST is being stopped:

$$\begin{array}{ll} & [true^* \cdot Start(ET) \cdot (\neg Done(ET))^* \cdot Resume(ST)]false \\ \wedge & [true^* \cdot Start(ET) \cdot (\neg Done(ET))^* \cdot Done(ST)]false \end{array}$$

The analysis shows that three of the eight safety rules are superfluous, i.e., a warning can not arise in any behaviour of the system (and hence do not occur if the controller implementation is without any flaws).

The second class of rules consists of 30 safety properties which only allow the execution of a task (T), if it is safe to do so. These tasks involve the movement of the printhead calibration system or shuttle. Since they physically operate in each other's workspace, it is possible that the system can incur physical damage if such a safety property is violated. As a result the system halts, if a rule is violated. To verify that the rules are valid throughout execution, temporal logic formulae of the conjunction of the following forms have been constructed (S, T, and U are actions),

$$[true^* \cdot S \cdot (\neg T)^* \cdot U]false$$

where T is the task that may not be executed between tasks S and U.

All of the formulae have been checked and four requirements are violated in the model. Since the verification has been performed with a controller that is in no way restricted, the controller should contain restricted behaviour (e.g., should never perform tasks in a certain order) in order to rule out the requirement violations.

Translating the code by hand to the model, has been achieved by taking incremental steps that add more and more involved C#-files until we obtained a model without any deadlocks. If a deadlock was encountered in the model, it meant that the model was incomplete, e.g., a communication between interfaces was missing.

In order to verify temporal formulae the specification is linearised to a linearised process specification. For all the requirements, this linearisation step has been executed just once. This took approximately 53 minutes on a computer with an Intel® Pentium® D930 processor and 2 Gb RAM running Linux. The subsequent verification for a single requirement took less than 15 minutes.

6 Related work

To determine if a system is free from programming bugs, inconsistencies, run-time errors, or non-portable constructs various tools like LINT [9, 17], POLYSPACE [16], and QA-C++ [24] can act as an extension to standard debuggers. When it comes to the verification of dynamic properties (deadlocks, unexpected behaviour) tools like Java PathFinder [23] or StEAM [20] can be used. These tools use a virtual machine in which models are translated to byte code, and executed afterwards to verify properties. Unfortunately, the size of the code is related to the underlying state space that needs to be explored, e.g., it becomes harder, or even impossible to verify dynamic safety properties. As stated by Java PathFinder: "While software model checking in theory sounds like a safe and robust verification method, reality shows that it does not scale well."

One can argue that the work presented here is comparable to the theory of abstract interpretation. In abstract interpretation [22], abstract values are chosen for variables. Behavioural models obtained via this approach depend on the (initial) values of data variables. Consequently, it requires manipulation of the data variables. For relatively small systems, this method is fruitful. However, for larger systems, this may lead to a state space explosion, due to the number of parallel processes combined with the number of possible abstract data values. In order to verify larger systems, either a more coarse grained abstraction is required (thereby losing information) as we do in this paper or state space reduction techniques (symmetry reduction, bisimulation reduction, etc.) need to be applied. Since almost every thread specifies unique behaviour, we could not benefit greatly from symmetry reduction. The application of bisimulation reduction techniques requires the generation of the underlying state space. Without any abstraction techniques, the approximated size of the state space should roughly be 10^{1000} states.

Work related to our method can also be found in the Bandera tool set [8]. The Bandera tool set translates Java source code to a model, which is used to verify properties about the system by model checking techniques. Unfortunately, large (software) systems lead to state spaces that are beyond today's computational power. The Bandera tool set itself, only accepts closed code. For

this reason the system needs to be complete before it can be verified. With help of extensions it is possible to verify open systems (e.g., an environment generator for Bandera [28]), but it still requires a full and correct implementation of a source code unit. Since our method abstracts from variables we can deal with partly implemented units and code skeletons.

The author of [18] presents a way for checking component behaviour compatibility, written in behaviour protocols and checked with the Spin model checker afterwards. Using LTL formulae, they manage to verify properties on a well documented system of 20 components. In our case study we tackled a bigger system running 230 concurrent processes, and performed a successful verification with a different tool set. Next to that the semantics of our components differs: we cope with processes that can be suspended and need to be resumed afterwards, while the components mentioned in [18] do not facilitate this mechanism.

Work presented in [15] shows a method for directly deriving a Promela specification from C code. This technique creates for every command a corresponding action in a Promela specification. In [31] another approach is taken with Promela. Here experiments are conducted with a virtual machine based approach for state space generation. By evaluating the byte-code language, they provide a way to efficiently execute operational semantics for modelling and programming languages. Undoubtedly, these techniques perform well on small toy examples for examining specific code constructs. However when changing the scope from specific code constructs to the control flow for examining larger concurrent systems, more rigorous techniques are required. In that sense, the method described in this paper can be viewed as an extension to their techniques.

Notice that our work shares resemblance with SLAM by Ball et al. [3]. One of the SLAM approaches is based on refining the abstractions (in order to rule out spurious counter-examples), and turns software implementations into boolean programs [2]. The basic idea is to leave out data initially, and include it when needed later on. Data that is included in the refinement applies to variables that are used in conditions. With help of a theorem prover and additional iterations for refinement the SLAM method tries to determine if it can solve the equations, thereby terminating the loop. In rare cases, it is possible that the theorem prover used by SLAM cannot solve the equations, which leads to a non-terminating algorithm. Consequently, verifying safety requirements becomes impossible. Our method does not use a theorem prover. If variables of a loop condition can change their values we assume that the condition eventually is violated, by which the loop terminates.

Counterexample-Guided Abstraction Refinement (CEGAR) (see [6]) is an automatic iterative abstraction-refinement methodology for which a datapath abstraction results in an approximation of the original design, i.e., if the approximation turns out to be too coarse, the approximation is automatically refined up to a point for which it can either generate a counter example or disprove it. While this technique is adaptive, our method is not. Therefore our approach can be seen as an instantiation of a first time right for CEGAR.

In D-Finder [4], a compositional method for checking invariance properties is presented. The basis of the method is an algorithm that iteratively computes invariants of components until they are strong enough to imply a global invariant that needs to be checked. In contrast with our method, where an over-approximation of the model is obtained, the method used in D-finder over-approximates the local properties of the components.

Another approach related to ours, can be found in VeriSoft [11]. Their approach consists of a systematic exploration of a state space by executing arbitrary code written in any language.

They guarantee complete state space coverage up to some depth, hence a partial state space exploration. Consequently, this only guarantees safety properties up to a certain depth/bound and not for the entire system.

Another related approach is program slicing [32]. This technique selects parts of the source code which are of interest to the values of specific variables. Our approach takes this to the extreme, by abstracting from all the variables and focus on calls between interfaces. Perhaps the technique of program slicing could also have been instrumental in abstracting from less relevant aspects of the model such as the logging of events.

7 Concluding remarks

In this paper, we have shown how safety properties can be verified for complex systems consisting of over more than 200 processes running in parallel. In particular, we have proposed a procedure for translating code specifying behaviour into mCRL2, with the help of an intermediate programming language preserving a simulation relation. Although we explicit mention the use of mCRL2 model checker, this method can also be incorporated by other model checkers.

The smallest state space that we were able to derive from the model consisted of 76256 unique states and 253145 transitions for 236 tasks running in parallel. The rather small amount of states results from the dependencies between the mutual nodes.

In this paper, the method is described in terms of a general programming language SCPL and modelling language mCRL2. In principle, the method can be used in combination with many implementation languages (C, C++, C#, Pascal, Delphi, Java ...) and verification languages that have similar constructs for describing behaviour such as synchronized communication, sorts to encode different processes and non-determinism. The semantics-preserving translation of a real-life programming language to mCRL2 is considered future work.

Another possibility for future work is to add resources in the model e.g., time to complete a task, power consumption or the network traffic weight. By adding these it is possible to check whether the model exceeds maximal outer limits or claim more resources than available.

Future research can be conducted in the area of model refinement. If a safety property does not hold, it might be that the property is violated as a result of the over-approximation. The same holds for a liveness property that is fulfilled. To determine if these are artifacts the relevant conditions and variables need to be incorporated in the model. While this is feasible to do by hand for small systems, it is impossible for industrial sized systems. Therefore automatic model refinement, may be included in future research activities.

Finally, we are considering to study the over-approximation technique in isolation. For this approach, existing mCRL2 models that depend on data and for which the requirements are expressed independently of the data may be considered. On such models we may conduct experiments that indicate whether applying the abstraction technique from this paper significantly reduces the number of states and whether this helps in verifying requirements.

Acknowledgements This work is supported as part of the ITEA project Twins 05004. We would like to thank NBG Industrial Automation for the opportunity to conduct the experiments on the Lunaris, Muck van Weerdenburg for his assistance in obtaining the temporal formulae,

and the reviewers for their constructive suggestions and valuable contributions.

Bibliography

[1] Tom Archer and Andrew Whitechapel. *Inside C#*. Pro-Developer Series. Microsoft Press, second edition edition, 2002.

[2] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification (SPIN'00), Stanford, CA, USA*, volume 1885 of *LNCS*, pages 113–130. Springer, 2000.

[3] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In Matthew B. Dwyer, editor, *Proceedings of the 8th international SPIN workshop on Model Checking Software (SPIN'01), Toronto, Ontario, Canada*, volume 2057 of *LNCS*, pages 103–122. Springer, 2001.

[4] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009), Grenoble, France*, volume 5643 of *LNCS*, pages 614–619. Springer, 2009.

[5] Julian Charles Bradfield. *Verifying Temporal Properties of Systems*. Progress in Theoretical Computer Science. Birkhäuser, 1992.

[6] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[8] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd international conference on Software engineering (ICSE 2000), Limerick, Ireland*, pages 439–448, 2000.

[9] Ian F. Darwin. *Checking C programs with Lint*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1988.

[10] G. Reza Djavanshir. Surveying the risks and benefits of it outsourcing. *IT Professional*, 7(6):32–37, 2005.

[11] Patrice Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 174–186. ACM Press, 1997.

[12] Jan Friso Groote and Radu Mateescu. Verification of temporal properties of processes in a setting with data. In Armando Martin Haeberer, editor, *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology (AMAST 1998), Amazonia, Brasil*, volume 1548 of *LNCS*, pages 74–90. Springer, 1999.

[13] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck van Weerdenburg. The formal specification language mCRL2. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[14] Jan Friso Groote, Aad H.J. Mathijssen, Michel A. Reniers, Yaroslav S. Usenko, and Muck J. van Weerdenburg. Analysis of distributed systems with mCRL2. In M. Alexander and W. Gardner, editors, *Process Algebra for Parallel and Distributed Processing*, chapter 4, pages 99–128. Taylor & Francis Group, 2009.

[15] Gerard J. Holzmann. From code to models. In *Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD 2001), Newcastle upon Tyne, UK*, pages 3–10. IEEE Computer Society Press, 2001.

[16] PolySpace Inc. Polyspace verification toolsuite. http://www.polyspace.com.

[17] Stephen Curtis Johnson. Lint, a C program checker. Technical Report Comp. Sci. Tech. Rep. 65, Bell Laboratories, 1978.

[18] Jan Kofron. Checking software component behavior using behavior protocols and SPIN. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC'07), Seoul, Korea*, pages 1513–1517. ACM Press, 2007.

[19] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[20] Peter Leven, Tilman Mehler, and Stefan Edelkamp. Directed error detection in C++ with the assembly-level model checker StEAM. In Susanne Graf and Laurent Mounier, editors, *Proceedings of the 11th International SPIN Workshop on Model Checking Software (SPIN), Barcelona, Spain*, volume 2989 of *LNCS*, pages 39–56. Springer, 2004.

[21] Sesh Murthy, Rama Akkiraju, Richard Goodwin, Pinar Keskinocak, John Rachlin, Frederick Wu, James Yeh, Robert Fuhrer, Santhosh Kumaran, Alok Aggarwal, Martin Sturzenbecker, Ranga Jayaraman, Robert Daigle, and Jan A. Van Mieghem. Coordinating investment, production, and subcontracting. *Manage. Sci.*, 45(7):954–971, 1999.

[22] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[23] Java PathFinder, August 2008. http://javapathfinder.sourceforge.net.

[24] PRQA. QA-C++ toolsuite. http://www.programmingresearch.com/.

[25] C.V. Ramamoorthy, C. Chandra, H.G. Kim, Y.C. Shim, and V. Vij. Systems integration: problems and approaches. In *Proceedings of the Second International Conference onSystems Integration (ICSI'92), Morristown, NJ, USA*, pages 522–529, 1992.

[26] Nieke Roos. Océ geeft aanzet tot open innovatie in inkjet, August 2007. Mechatronica Magazine.

[27] D.P. Siewiorek, R. Chillarege, and Z.T. Kalbarczyk. Reflections on industry trends and experimental research in dependability. *IEEE Transactions on Dependable and Secure Computing*, 1(2):109–127, April-June 2004.

[28] Oksana Tkachuk, Matthew B. Dwyer, and Corina S. Pasareanu. Automated environment generation for software model checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003), Montreal, Canada*, pages 116–129. IEEE Computer Society Press, 2003.

[29] Rob J. van Glabbeek and Ursula Goltz. Equivalence notions for concurrent systems and refinement of actions (extended abstract). In Antoni Kreczmar and Grazyna Mirkowska, editors, *Proceedings of Mathematical Foundations of Computer Science 1989 (MFCS'89), Porabka-Kozubnik, Poland*, volume 379 of *LNCS*, pages 237–248. Springer, 1989.

[30] Joost van Lier, Inneke Van Nieuwenhuyse, Liesje De Boeck, Ton Dohmen, Nico Vandaele, and Marc Lambrecht. Benefits management and strategic alignment in an IT outsourcing context. In *Proceedings of the 40th Hawaii International International Conference on Systems Science (HICSS-40 2007), Waikoloa, Big Island, HI, USA*. IEEE Computer Society Press, 2007.

[31] M. Weber. An embeddable virtual machine for state space generation. In D. Bosnacki and S. Edelkamp, editors, *Proceedings of the 14th International SPIN Workshop on Model Checking Software (SPIN), Berlin, Germany*, volume 4595 of *LNCS*, pages 168–186. Springer, 2007.

[32] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE'81), San Diego, CA, USA*, pages 439–449. IEEE Computer Society Press, 1981.